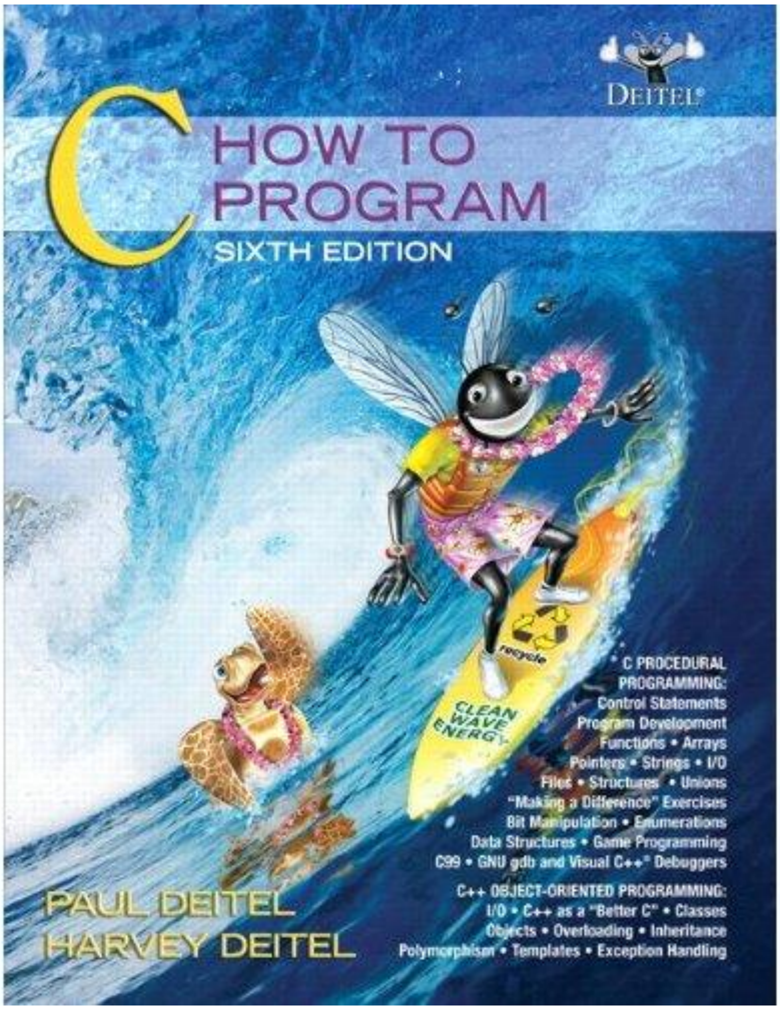


Como programar Em C

Paul J. Deitel e Harvey M. Deitel

Índice

1

Conceitos de Computação

Objetivos

- Entender os conceitos básicos do computador.
- Familiarizar-se com os diferentes tipos de linguagens de programação.
- Familiarizar-se com a história da linguagem de programação C.
- Conhecer Biblioteca Padrão da linguagem C (C Standard Library).
- Entender o ambiente e desenvolvimento de programas C.
- Compreender por que é apropriado aprender C no primeiro curso de programação.
- Compreender por que a linguagem C fornece uma base para estudos futuros de programação em geral e em particular para o C++.

As coisas são sempre melhores no começo.
Blaise Pascal

Grandes pensamentos exigem grandes linguagens.
Aristófanes

Nossa vida é desperdiçada em detalhes. Simplifique, simplifique.
Henry Thoreau

Sumário

1.1 Introdução

Bem-vindo ao C! Trabalhamos muito duro para criar o que sinceramente esperamos ser uma maneira instrutiva e divertida de aprendizado. O C é uma linguagem difícil, normalmente ensinada apenas para os programadores experientes, e, sendo assim, este livro não tem similar entre os livros-texto de C:

Este livro é aconselhável para pessoas interessadas em aspectos técnicos e com pouca ou nenhuma experiência de programação.

Este livro também é aconselhável para programadores experientes que desejam um tratamento profundo e rigoroso da linguagem.

Como um livro pode despertar o interesse de ambos os grupos? A resposta é que o tema central do livro coloca em destaque a obtenção de *clareza* nos programas através de técnicas comprovadas de programação estruturada. Quem não é programador aprenderá a programar "certo" desde o início. Tentamos escrever de uma maneira clara e simples. O livro contém muitas ilustrações. Talvez o aspecto mais importante seja o de que o livro apresenta um grande número de programas práticos em C e mostra as saídas produzidas quando eles forem executados em um computador.

Os quatro primeiros capítulos apresentam os fundamentos da computação, da programação de computadores e da linguagem de programação C. As análises estão inseridas em uma introdução à programação de computadores usando um método estruturado. Os principiantes em programação que fizeram nossos cursos nos informaram que o material desses capítulos apresenta uma base sólida para as técnicas mais avançadas da linguagem C. Normalmente os programadores experientes lêem rapidamente os quatro primeiros capítulos e então descobrem que o modo com o qual o assunto é tratado no Capítulo 5 é rigoroso e fascinante. Eles gostam particularmente da maneira detalhada como são analisados os ponteiros, strings, arquivos e estruturas de dados nos capítulos que se seguem.

Muitos programadores experientes nos disseram que aprovam nosso modo de apresentar a programação estruturada. Freqüentemente eles costumam programar em uma linguagem estruturada como o Pascal, mas, por nunca terem sido apresentados formalmente à programação estruturada, não escrevem o melhor código possível. À medida que aprenderem C com este livro, eles poderão aprimorar seu estilo de programação. Dessa forma, quer você seja um principiante, quer seja um programador experiente, há muito aqui para informá-lo, diverti-lo e estimulá-lo.

A maioria das pessoas está familiarizada com as coisas excitantes que os computadores fazem. Neste curso, você aprenderá a mandar os computadores fazerem essas coisas. É o *software* (i.e., as instruções escritas para mandar o computador realizar ações e tomar decisões) que controla os computadores (chamados freqüentemente de *hardware),* e uma das linguagens de desenvolvimento de software mais populares atualmente é o C. Este texto fornece uma introdução à programação em ANSI C, a versão padronizada em 1989 tanto nos EUA, através do American National Standards Institute (ANSI), como em todo o mundo, através da International Standards Organization (ISO).

O uso de computadores está aumentando em quase todos os campos de trabalho. Em uma era na qual os custos crescem constantemente, os custos com a computação diminuíram drasticamente devido aos excitantes desenvolvimentos em tecnologia de software e hardware. Os computadores que podem ter ocupado salas enormes e custado milhões de dólares 25 anos atrás agora podem estar gravados na superfície de chips de silício menores do que uma unha e que talvez custem alguns dólares cada. Ironicamente, o silício é um dos materiais mais abundantes na Terra — ele é um componente da areia comum. A tecnologia do chip de silício tornou a computação tão econômica que aproximadamente 150 milhões de computadores de uso geral estão sendo empregados em todo o mundo, ajudando as pessoas no comércio, indústria, governo e em suas vidas particulares. Este número pode dobrar facilmente em alguns anos.

A linguagem C pode ser ensinada em um primeiro curso de programação, o público pretendido por este livro? Pensamos que sim. Dois anos atrás, aceitamos este desafio quando o Pascal estava solida mente estabelecido como a linguagem dos primeiros cursos de ciência da computação. Escrevemos *Como Programar em C*, a primeira edição deste livro. Centenas de universidades em todo o mundo usaram *Como Programar em C.* Cursos baseados nesse livro se provaram tão eficientes quanto seus predecessores baseados no Pascal. Não foram observadas diferenças significativas, exceto que os alunos estavam mais motivados por saberem que era mais provável usar o C do que o Pascal em seus cursos mais avançados e em suas carreiras. Os alunos que aprendem o C também sabem que estarão mais preparados para aprender o C+ + mais rapidamente. O C+ + é um superconjunto da linguagem C que se destina aos programadores que desejam escrever programas orientados a objetos. Falaremos mais sobre o C++ na Seção 1.14.

Na realidade, o C + + está recebendo tanto interesse que incluímos uma introdução detalhada ao C+ + e programação orientada a objetos. Um fenômeno interessante que ocorre no campo das linguagens de programação é que atualmente muitos dos revendedores principais simplesmente comercializam um produto que combina C/C++, em vez de oferecer produtos separados. Isto dá aos usuários a capacidade de continuar programando em C se desejarem e depois migrarem gradualmente para o C ++ quando acharem apropriado.

Agora você já sabe de tudo! Você está prestes a iniciar uma jornada fascinante e gratificadora. À medida que for em frente, se você desejar entrar em contato conosco, envie-nos um e-mail para **deitel@world.std.com** pela Internet. Envidaremos todos os esforços para oferecer uma resposta rápida. Boa sorte!

1.2 O que É um Computador?

Um *computador* é um dispositivo capaz de realizar cálculos e tomar decisões lógicas com uma velocidade milhões ou mesmo bilhões de vezes mais rápida do que os seres humanos. Por exemplo, muitos dos computadores pessoais de hoje podem realizar dezenas de milhões de operações aritméticas por segundo. Uma pessoa utilizando uma calculadora de mesa poderia levar décadas para realizar o mesmo número de operações que um poderoso computador pessoal pode realizar em segundos. (Aspectos para reflexão: Como você saberia se a pessoa fez as operações corretamente? Como você saberia se o computador fez as operações corretamente?) Os mais rápidos *supercomputadores* de hoje podem realizar centenas de bilhões de operações por segundo — quase tantas operações quanto centenas de pessoas poderiam realizar em um ano! E os computadores que permitem trilhões de instruções por segundo já se encontram em funcionamento em laboratórios de pesquisa.

Os computadores processam *dados* sob o controle de conjuntos de instruções chamados *programas de computador.* Estes programas conduzem o computador através de um conjunto ordenado de ações especificado por pessoas chamadas *programadores de computador.*

Os vários dispositivos (como teclado, tela, discos, memória e unidades de processamento) que constituem um sistema computacional são chamados de *hardware.* Os programas executados em um computador são chamados de *software.* O custo do hardware diminuiu drasticamente nos últimos anos, chegando ao ponto de os computadores pessoais se tornarem uma utilidade. Infelizmente, o custo do desenvolvimento de software tem crescido constantemente, à medida que os programadores desenvolvem aplicações cada vez mais poderosas e complexas, sem serem capazes de fazer as melhorias correspondentes na tecnologia necessária para esse desenvolvimento. Neste livro você aprenderá métodos de desenvolvimento de software que podem reduzir substancialmente seus custos e acelerar o processo de desenvolvimento de aplicações poderosas e de alto nível. Estes métodos incluem *programação estruturada, refinamento passo a passo top-down {descendente), funcionalização* e, finalmente, *programação orientada a objetos.*

1.3 Organização dos Computadores

Independentemente das diferenças no aspecto físico, praticamente todos os computadores podem ser considerados como divididos em seis *unidades lógicas* ou seções. São elas:

1. ***Unidade de entrada (input unit).*** Esta é a seção de "recepção" do computador. Ela obtém as informações (dados e programas de computador) dos vários *dispositivos de entrada (input devices)* e as coloca à disposição de outras unidades para que possam ser processadas. A maior parte das informações é fornecida aos computadores atualmente através de teclados como os de máquinas de escrever.

2. ***Unidade de saída (output unit).*** Esta é a seção de "expedição" do computador. Ela leva as informações que foram processadas pelo computador e as envia aos vários *dispositivos de saída (output devices)* para torná-las disponíveis para o uso no ambiente externo ao computador. A maioria das informações é fornecida pelo computador através de exibição na tela ou impressão em papel.

3. ***Unidade de memória (memory unit).*** Este é a seção de "armazenamento" do computador, com acesso rápido e capacidade relativamente baixa. Ela conserva as informações que foram fornecidas através da unidade de entrada para que possam estar imediatamente disponíveis para o processamento quando se fizer necessário. A unidade de memória também conserva as informações que já foram processadas até que sejam enviadas para os dispositivos de saída pela unidade de saída. Freqüentemente a unidade de memória é chamada de *memória (memory), memória principal* ou *memória primária (primary memory).*

4. ***Unidade aritmética e lógica (arithmetic and logic unit, ALU).*** Esta é a seção de "fabricação" do computador. Ela é a responsável pela realização dos cálculos como adição, subtração, multiplicação e divisão. Ela contém os mecanismos de decisão que permitem ao computador, por exemplo, comparar dois itens da unidade de memória para determinar se são iguais ou não.

5. ***Unidade central de processamento, UCP (central processing unit, CPU).*** Esta é a seção "administrativa" do computador. Ela é o coordenador do computador e o responsável pela supervisão do funcionamento das outras seções. A CPU informa à unidade de entrada quando as informações devem ser lidas na unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas em cálculos e informa à unidade de saída quando as informações devem ser enviadas da unidade de memória para determinados dispositivos de saída.

6. ***Unidade de memória secundária (secondary storage unit).*** Esta é a seção de "armazenamento" de alta capacidade e de longo prazo do computador. Os programas ou dados que não estiverem sendo usados ativamente por outras unidades são colocados normalmente em dispositivos de memória secundária (como discos) até que sejam outra vez necessários, possivelmente horas, dias, meses ou até mesmo anos mais tarde.

1.4 Processamento em Lotes (Batch Processing), Multiprogramação e Tempo Compartilhado (Timesharing)

Os primeiros computadores eram capazes de realizar apenas um trabalho ou tarefa de cada vez. Esta forma de funcionamento de computadores é chamada freqüentemente de processamento em lotes de usuário único. O computador executa um único programa de cada vez enquanto processa dados em gru¬pos ou lotes (batches). Nesses primeiros sistemas, geralmente os usuários enviavam suas tarefas ao centro computacional em pilhas de cartões perfurados. Freqüentemente os usuários precisavam esperar horas antes que as saídas impressas fossem levadas para seus locais de trabalho.

À medida que os computadores se tornaram mais poderosos, tornou-se evidente que o processamen¬to em lotes de usuário único raramente utilizava com eficiência os recursos do computador. Em vez disso, imaginava-se que muitos trabalhos ou tarefas poderiam ser executados de modo a compartilhar os re¬cursos do computador e assim conseguir utilizá-lo melhor. Isto é chamado multiprogramação. A multiprogramação envolve as "operações" simultâneas de muitas tarefas do computador — o computa¬dor compartilha seus recursos entre as tarefas que exigem sua atenção. Com os primeiros sistemas de multiprogramação, os usuários ainda enviavam seus programas em pilhas de cartões perfurados e espe¬ravam horas ou dias para obter os resultados.

Nos anos 60, vários grupos de indústrias e universidades foram pioneiros na utilização do conceito de timesharing (tempo compartilhado). Timesharing é um caso especial de multiprogramação no qual os usuários podem ter acesso ao computador através de dispositivos de entrada/saída ou terminais. Em um sistema computacional típico de timesharing, pode haver dezenas de usuários compartilhando o computador ao mesmo tempo. Na realidade o computador não atende a todos os usuários simultanea¬mente. Em vez disso, ele executa uma pequena parte da tarefa de um usuário e então passa a fazer a tarefa do próximo usuário. O computador faz isto tão rapidamente que pode executar o serviço de cada usuário várias vezes por segundo. Assim, parece que as tarefas dos usuários estão sendo executadas simultaneamente.

1.5 Computação Pessoal, Computação Distribuída e Computação Cliente/Servidor

Em 1997, a Apple Computer tornou popular o fenômeno da computação pessoal. Inicialmente, isto era um sonho de quem a tinha como um *hobby.* Computadores tornaram-se suficientemente baratos para serem comprados para uso pessoal ou comercial. Em 1981, a IBM, a maior vendedora de computadores do mundo, criou o IBM PC (Personal Computer, computador pessoal). Do dia para a noite, literalmente, a computação pessoal se tornou comum no comércio, na indústria e em organizações governamentais.

Mas esses computadores eram unidades "autônomas" — as pessoas faziam suas tarefas em seus próprios equipamentos e então transportavam os discos de um lado para outro para compartilhar as informações. Embora os primeiros computadores pessoais não fossem suficientemente poderosos para serem compartilhados por vários usuários, esses equipamentos podiam ser ligados entre si em redes de computadores, algumas vezes através de linhas telefônicas e algumas vezes em redes locais de organizações. Isto levou ao fenômeno da *computação distribuída,* na qual a carga de trabalho computacional de uma organização, em vez de ser realizada exclusivamente em uma instalação central de informática, é distribuída em redes para os locais (sites) nos quais o trabalho real da organização é efetuado. Os computadores pessoais eram suficientemente poderosos para manipular as exigências computacionais de cada usuário em particular e as tarefas básicas de comunicações de passar as informações eletronicamente de um lugar para outro.

Os computadores pessoais mais poderosos de hoje são tão poderosos quanto os equipamentos de milhões de dólares de apenas uma década atrás. Os equipamentos desktop (computadores de mesa) mais poderosos — chamados *workstations* ou *estações de trabalho* — fornecem capacidades enormes a usuários isolados. As informações são compartilhadas facilmente em redes de computadores onde alguns deles, os chamados *servidores de arquivos (file servers),* oferecem um depósito comum de programas e dados que podem ser usados pelos computadores *clientes (clients)* distribuídos ao longo da rede, daí o termo *computação cliente/servidor.* O C e o C ++ tornaram-se as linguagens preferidas de programação para a criação de software destinado a sistemas operacionais, redes de computadores e aplicações distribuídas cliente/servidor.

1.6 Linguagens de Máquina, Linguagens Assembly e Linguagens de Alto Nível

Os programadores escrevem instruções em várias linguagens de programação, algumas entendidas diretamente pelo computador e outras que exigem passos intermediários de *tradução.* Centenas de linguagens computacionais estão atualmente em uso. Estas podem ser divididas em três tipos gerais:

1. Linguagens de máquina
2. Linguagens assembly
3. Linguagens de alto nível

Qualquer computador pode entender apenas sua própria *linguagem de máquina.* A linguagem de máquina é a "linguagem natural" de um determinado computador. Ela está relacionada intimamente com o projeto de hardware daquele computador. Geralmente as linguagens de máquina consistem em strings de números (reduzidos em última análise a ls e Os) que dizem ao computador para realizar uma de suas operações mais elementares de cada vez. As linguagens de máquina são *dependentes de máquina (não-padronizadas,* ou *machine dependent),* i.e., uma determinada linguagem de máquina só pode ser usada com um tipo de computador. As linguagens de máquina são complicadas para os humanos, como se pode ver no trecho seguinte de um programa em linguagem de máquina que adiciona o pagamento de horas extras ao salário base e armazena o resultado no pagamento bruto.

+1300042774
+1400593419
+1200274027

À medida que os computadores se tornaram mais populares, ficou claro que a programação em linguagem de máquina era simplesmente muito lenta e tediosa para a maioria dos programadores. Em vez de usar strings de números que os computadores podiam entender diretamente, os programadores começaram a usar abreviações parecidas com palavras em inglês para representar as operações elementares de um computador. Estas abreviações formaram a base das *linguagens assembly.* Foram desenvolvidos *programas tradutores,* chamados *assemblers,* para converter programas em linguagem assembly para linguagem de máquina na velocidade ditada pelo computador. O trecho de um programa em linguagem assembly a seguir também soma o pagamento de horas extras ao salário base e armazena o resultado em pagamento bruto, porém isto é feito de uma forma mais clara do que o programa equivalente em linguagem de máquina.

LOAD BASE
ADD EXTRA
STORE BRUTO

O uso do computador aumentou rapidamente com o advento das linguagens assembly, mas elas ainda exigiam muitas instruções para realizar mesmo as tarefas mais simples. Para acelerar o processo de programação, foram desenvolvidas *linguagens de alto nível,* nas quais podiam ser escritas instruções simples para realizar tarefas

fundamentais. Os programas tradutores que convertiam programas de linguagem de alto nível em linguagem de máquina são chamados *compiladores.* As linguagens de alto nível permitem aos programadores escrever instruções que se parecem com o idioma inglês comum e contêm as notações matemáticas normalmente usadas. Um programa de folha de pagamento em uma linguagem de alto nível poderia conter uma instrução como esta:

Bruto = Base + Extra

Obviamente, as linguagens de alto nível são muito mais desejáveis do ponto de vista do programador do que as linguagens de máquina ou assembly. O C e o C++ estão entre as linguagens de alto nível mais poderosas e mais amplamente usadas.

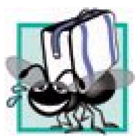
1.7 A História do C

O C foi desenvolvido a partir de duas linguagens anteriores, o BCPL e o B. O BCPL foi desenvolvido em 1967 por Martin Richards como uma linguagem para escrever software de sistemas operacionais e compiladores. Ken Thompson modelou muitos recursos em sua linguagem B com base em recursos similares do BCPL e usou o B para criar as primeiras versões do sistema operacional UNIX nas instalações do Bell Laboratories em 1970, em um computador DEC PDP-7. Tanto o BCPL como o B eram linguagens "sem tipos" ("typeless") — todos os itens de dados ocupavam uma "palavra" na memória e a responsabilidade de lidar com um item de dados como um número inteiro ou real, por exemplo, recaía sobre os ombros do programador.

A linguagem C foi desenvolvida a partir do B por Dennis Ritchie, do Bell Laboratories, e implementada originalmente em um computador DEC PDP-11, em 1972. De início o C se tornou amplamente conhecido como a linguagem de desenvolvimento do sistema operacional UNIX. Hoje em dia, praticamente todos os grandes sistemas operacionais estão escritos em C e/ou C++ . Ao longo das duas últimas décadas, o C ficou disponível para a maioria dos computadores. O C independe do hardware. Elaborando um projeto cuidadoso, é possível escrever programas em C que sejam portáteis para a maioria dos computadores. O C usa muitos dos importantes conceitos do BCPL e do B ao mesmo tempo que adiciona tipos de dados e outros recursos poderosos.

No final da década de 70, o C evoluiu e chegou ao que se chama agora de "C tradicional". A publicação em 1978 do livro de Kernighan e Ritchie, *The C Programming Language,* fez com que essa linguagem recebesse muita atenção. Esta publicação se tornou um dos livros de informática mais bem-sucedidos de todos os tempos.

A rápida expansão do C em vários tipos de computadores (algumas vezes chamados de *plataformas de hardware)* levou a muitas variantes. Elas eram similares, mas freqüentemente incompatíveis. Isto foi um problema sério para os desenvolvedores de programas que precisavam criar um código que fosse executado em várias plataformas. Ficou claro que era necessário uma versão padrão do C. Em 1983, foi criado o comitê técnico X3J11 sob o American National Standards Committee on Computers and Information Processing (X3) para "fornecer à linguagem uma definição inequívoca e independente de equipamento". Em 1989, o padrão foi aprovado. O documento é conhecido como ANSI/ISO 9899:1990. Pode-se pedir cópias desse documento para o American National Standards Institute, cujo endereço consta no Prefácio a este texto. A segunda edição do livro de Kernighan e Ritchie, publicada em 1988, reflete esta versão, chamada ANSI C, agora usada em todo o mundo (Ke88).



Dicas de portabilidade 1.1

Como o C é uma linguagem independente de hardware e amplamente disponível, as aplicações escritas em C podem ser executadas com pouca ou nenhuma modificação em uma grande variedade de sistemas computacionais.

1.8 A Biblioteca Padrão (Standard Library) do C

Como você aprenderá no Capítulo 5, os programas em C consistem em módulos ou elementos chamados *funções.* Você pode programar todas as funções de que precisa para formar um programa C, mas a maioria dos programadores C tira proveito de um excelente conjunto de funções chamado *C Standard Library* (Biblioteca Padrão do C). Dessa forma, há na realidade duas partes a serem aprendidas no "mundo" do C. A primeira é a linguagem C em si, e a segunda é como usar as funções do C Standard Library. Ao longo deste livro, analisaremos muitas dessas funções. O Apêndice A (condensado e adaptado do documento padrão do ANSI C) relaciona todas as funções disponíveis na biblioteca padrão do C. A leitura do livro de Plauger (P192) é obrigatória para os programadores que necessitam de um entendimento profundo das funções da biblioteca, de como implementá-las e como usá-las para escrever códigos portáteis.

Neste curso você será estimulado a usar o *método dos blocos de construção* para criar programas. Evite reinventar a roda. Use os elementos existentes — isto é chamado *reutilização de software* e é o segredo do campo de desenvolvimento da programação orientada a objetos. Ao programar em C, você usará normalmente os seguintes blocos de construção:

- Funções da C Standard Library (biblioteca padrão)
- Funções criadas por você mesmo
- Funções criadas por outras pessoas e colocadas à sua disposição

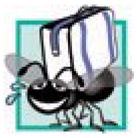
A vantagem de criar suas próprias funções é que você saberá exatamente como elas funcionam. Você poderá examinar o código C. A desvantagem é o esforço demorado que se faz necessário para projetar e desenvolver novas funções.

Usar funções existentes evita reinventar a roda. No caso das funções standard do ANSI, você sabe que elas foram desenvolvidas cuidadosamente e sabe que, por estar usando funções disponíveis em praticamente todas as implementações do ANSI C, seus programas terão uma grande possibilidade de serem portáteis.



Dica de desempenho 1.1

Usar as funções da biblioteca standard do C em vez de você escrever suas próprias versões similares pode melho¬rar o desempenho do programa porque essas funções foram desenvolvidas cuidadosamente por pessoal eficiente.



Dicas de portabilidade 1.2

Usar as funções da biblioteca padrão do C em vez de escrever suas próprias versões similares pode melhorar a porta¬bilidade do programa porque essas funções estão colocadas em praticamente todas as implementações do ANSI C.

1.9 Outras Linguagens de Alto Nível

Centenas de linguagens de alto nível foram desenvolvidas, mas apenas algumas conseguiram grande aceitação. O *FORTRAN* (FORmula TRANslator) foi desenvolvido pela IBM entre 1954 e 1957 para ser usado em aplicações científicas e de engenharia que exigem cálculos matemáticos complexos. O FORTRAN ainda é muito usado.

O *COBOL* (COmmon Business Oriented Language) foi desenvolvido em 1959 por um grupo de fabricantes de computadores e usuários governamentais e industriais. O COBOL é usado principalmente para aplicações comerciais que necessitam de uma manipulação precisa e eficiente de grandes volumes de dados. Hoje em dia, mais de metade de todo o software comercial ainda é programada em COBOL. Mais de um milhão de pessoas estão empregadas como programadores de COBOL.

O *Pascal* foi desenvolvido quase ao mesmo tempo que o C. Ele destinava-se ao uso acadêmico. Falaremos mais sobre o Pascal na próxima seção.

1.10 Programação Estruturada

Durante os anos 60, enormes esforços para o desenvolvimento de software encontraram grandes dificuldades. Os cronogramas de desenvolvimento de software normalmente estavam atrasados, os custos superavam em muito os orçamentos e os produtos finais não eram confiáveis. As pessoas começaram a perceber que o desenvolvimento era uma atividade muito mais complexa do que haviam imaginado. A atividade de pesquisa dos anos 60 resultou na evolução da *programação estruturada* — um método disciplinado de escrever programas claros, nitidamente corretos e fáceis de modificar. Os Capítulos 3 e 4 descrevem os fundamentos da programação estruturada. O restante do texto analisa o desenvolvimento de programas em C estruturados.

Um dos resultados mais tangíveis dessa pesquisa foi o desenvolvimento da linguagem Pascal de programação pelo Professor Nicklaus Wirth em 1971. O Pascal, que recebeu este nome em homenagem ao matemático e filósofo Blaise Pascal, que viveu no século XVII, destinava-se ao ensino da programação estruturada em ambientes acadêmicos e se tornou rapidamente a linguagem preferida para a introdução à programação em muitas universidades. Infelizmente, a linguagem carecia de muitos recursos necessários para torná-la útil em aplicações comerciais, industriais e governamentais, e portanto não foi amplamente aceita nesses ambientes. Possivelmente a história registra que a grande importância do Pascal foi sua escolha para servir de base para a linguagem de programação *Ada.*

A linguagem Ada foi desenvolvida sob a responsabilidade do Departamento de Defesa dos EUA (United States Department of Defense, ou DOD) durante os anos 70 e início dos anos 80. Centenas de linguagens diferentes estavam sendo usadas para produzir os imensos sistemas de comando e controle de software do DOD. O DOD desejava uma única linguagem que pudesse atender a suas necessidades. O Pascal foi escolhido como base, mas a linguagem Ada final é muito diferente do Pascal. A linguagem Ada recebeu este nome em homenagem a Lady Ada Lovelace, filha do poeta Lorde Byron. De uma maneira geral, Lady Lovelace é considerada a primeira pessoa do mundo a escrever um programa de computador, no início do século XIX. Uma característica importante da linguagem Ada é chamada *multitarefa (multitasking);* isto permite aos programadores especificarem a ocorrência simultânea de muitas atividades. Outras linguagens de alto nível amplamente usadas que analisamos — incluindo o C e o C++ — permitem ao programador escrever programas que realizem apenas uma atividade. Saberemos no futuro se a linguagem Ada conseguiu atingir seus objetivos de produzir software confiável e reduzir substancialmente os custos de desenvolvimento e manutenção de software.

1.11 Os Fundamentos do Ambiente C

Todos os sistemas C são constituídos geralmente de três partes: o ambiente, a linguagem e a C Standard Library. A análise a seguir explica o ambiente típico de desenvolvimento do C, mostrado na Figura 1.1.

Normalmente os programas em C passam por seis fases para serem executados (Figura 1.1). São elas: *edição, pré-processamento, compilação, linking (ligação), carregamento* e *execução.* Concentrar-nos-emos aqui em um sistema típico do C baseado em UNIX. Se você não estiver usando um sistema UNIX, consulte o manual de seu sistema ou pergunte ao seu professor como realizar estas tarefas em seu ambiente.

A primeira fase consiste na edição de um arquivo. Isto é realizado com um *programa editor.* O programador digita um programa em C com o editor e faz as correções necessárias. O programa é então armazenado em um dispositivo de armazenamento secundário como um disco. Os nomes de programas em C devem ter a extensão **.c.** Dois editores muito usados em sistemas UNIX são o **vi** e o **emacs.** Os pacotes de software C/C++ como o Borland C++ para IBM PCs e compatíveis e o Symantec C++ para Apple Macintosh possuem editores embutidos que se adaptam perfeitamente ao ambiente de programação. Partimos do princípio de que o leitor sabe editar um programa.

A seguir, o programador emite o comando de *compilar* o programa. O compilador traduz o programa em C para o código de linguagem de máquina (também chamado de *código-objeto).* Em um sistema C, um programa *pré-processador é* executado automaticamente antes de a fase de tradução começar. O pré-processador C obedece a comandos especiais chamados *diretivas do pré-processador* que indicam que devem ser realizadas determinadas manipulações no programa antes da compilação. Estas manipulações consistem normalmente em incluir outros arquivos no arquivo a ser compilado e substituir símbolos especiais por texto de programa. As diretivas mais comuns do pré-processador são analisadas nos primeiros capítulos; uma análise detalhada de todos os recursos do pré-processador está presente no Capítulo 13. O pré-processador é ativado automaticamente pelo compilador antes de o programa ser convertido para linguagem de máquina.

A quarta fase é chamada *linking.* Normalmente os programas em C contêm referências a funções definidas em outros locais, como nas bibliotecas padrão ou nas bibliotecas de um grupo de programadores que estejam trabalhando em um determinado projeto. Assim, o código-objeto produzido pelo compilador C contém normalmente "lacunas" devido à falta dessas funções. Um *linker* faz a ligação do código-objeto com o código das funções que estão faltando para produzir uma *imagem executável* (sem a falta de qualquer parte). Em um sistema típico baseado em UNIX, o comando para compilar e linkar um programa é **cc.** Por exemplo, para compilar e linkar um programa chamado **bemvindo. c** digite

cc bemvindo.c

no prompt do UNIX e pressione a tecla Return (ou Enter). Se o programa for compilado e linkado corretamente, será produzido um arquivo chamado **a.out.** Este arquivo é a imagem executável de nosso programa **bemvindo. c.**
A quinta fase é chamada *carregamento.* Um programa deve ser colocado na memória antes que possa ser executado pela primeira vez. Isto é feito pelo *carregador (rotina de carga ou loader),* que apanha a imagem executável do disco e a transfere para a memória.

Finalmente, o computador, sob o controle de sua CPU, executa as instruções do programa, uma após a outra. Para carregar e executar o programa em um sistema UNIX digitamos **a.out** no prompt do UNIX e apertamos a tecla Return.

A maioria dos programas em C recebe ou envia dados. Determinadas funções do C recebem seus dados de entrada a partir do ***stdin*** (o *dispositivo padrão de entrada,* ou *standard input device)* que normalmente é definido como o teclado, mas que pode estar associado a outro dispositivo. Os dados são enviados para o **stdout** (o *dispositivo padrão de saída,* ou *standard output device),* que normalmente é a tela do computador, mas que pode estar associado a outro dispositivo. Quando dizemos que um programa fornece um resultado, normalmente queremos dizer que o resultado é exibido na tela.

Há também um *dispositivo padrão de erros (standard error device)* chamado ***stderr.*** O dispositivo **stderr** (normalmente associado à tela) é usado para exibir mensagens de erro. É comum não enviar os dados regu-lares de saída, i.e., **stdout** para a tela e manter **stderr** associado a ela para que o usuário possa ser informado imediatamente dos erros.

1.12 Observações Gerais sobre o C e Este Livro

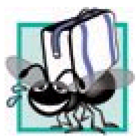
O C é uma linguagem difícil. Algumas vezes, os programadores experientes ficam orgulhosos de criar utilizações estranhas, distorcidas e complicadas da linguagem. Isto é uma péssima regra de programação. Ela faz com que os programas fiquem difíceis de ler, com grande probabilidade de se comportarem de maneira imprevista e mais difíceis de testar e depurar erros. Este livro se destina a programadores iniciantes, portanto damos ênfase à elaboração de programas claros e bem-estruturados. Um de nossos objetivos principais neste livro é fazer com que os programas fiquem *claros* através da utilização de técnicas comprovadas de programação estruturada e das muitas práticas recomendáveis, de programarão mencionadas.



Boas práticas de programação 1.1

Escreva seus programas em C de uma maneira simples e objetiva. Algumas vezes isto é chamado KIS (do inglês "keep it simple" [que pode ser traduzido por "mantenha a simplicidade"]). Não "complique" a linguagem tentando soluções "estranhas".

Você pode ouvir que o C é uma linguagem portátil e que os programas escritos em C podem ser executados em muitos computadores diferentes. *A portabilidade é um objetivo ilusório.* O documento padrão do ANSI (An90) lista 11 páginas de questões delicadas sobre portabilidade. Foram escritos livros completos sobre o assunto de portabilidade no C (Ja89) (Ra90).

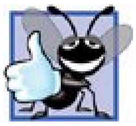


Dicas de portabilidade 1.3

Embora seja possível escrever programas portáteis, há muitos problemas entre as diferentes implementa¬ções do C e os diferentes computadores que tornam a portabilidade um objetivo difícil de atingir. Simples¬mente escrever programas em C não garante a portabilidade.

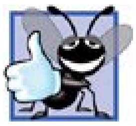
Fizemos uma pesquisa cuidadosa do documento padrão do ANSI C e examinamos nossa apresentação quanto aos aspectos de completude e precisão. Entretanto, o C é uma linguagem muito rica e possui algumas sutilezas e alguns assuntos avançados que não analisamos. Se você precisar conhecer detalhes técnicos adicionais sobre o ANSI C, sugerimos a leitura do próprio documento padrão do ANSI C ou o manual de referência de Kernighan e Ritchie (Ke88).

Limitamos nosso estudo ao ANSI C. Muitos recursos do ANSI C não são compatíveis com implementações antigas do C, portanto você pode vir a descobrir que alguns programas mencionados neste texto não funcionam com compiladores antigos do C.



Boas práticas de programação 1.2

Leia os manuais da versão do C que estiver usando. Consulte freqüentemente estes manuais para se certificar do conhecimento do rico conjunto de recursos do C e de que eles estão sendo usados corretamente.



Boas práticas de programação 1.3

Seu computador e compilador são bons mestres. Se você não estiver certo de como funciona um recurso do C, escreva um programa de teste que utilize aquele recurso, compile e execute o programa, e veja o que acontece.

1.13 Concurrent C

Foram desenvolvidas outras versões do C através de um esforço contínuo de pesquisa no Bell Laboratories. Gehani (Ge89) desenvolveu o *Concurrent C* — um superconjunto do C que inclui recursos para especificar a ocorrência de diversas atividades em paralelo. Linguagens como o Concurrent C e recursos de sistemas operacionais que suportam paralelismo em aplicações do usuário se tornarão cada vez mais populares na próxima década, à medida que o uso de *multiprocessadores* (i.e., computadores com mais de uma CPU) aumentar. Normalmente cursos e livros-texto de sistemas operacionais (De90) tratam do assunto de programação paralela de uma maneira consistente.

1.14 Programação Orientada a Objetos e C+ +

Outro superconjunto do C, especificamente o C++, foi desenvolvido por Stroustrup (St86) no Bell Laboratories. O C++ fornece muitos recursos que tornam a linguagem C mais "atraente". Porém o mais importante é que ela fornece recursos para a *programação orientada a objetos.*

Objetos são basicamente *componentes* reutilizáveis de software que modelam itens do mundo real. Está ocorrendo uma revolução na comunidade de software. Desenvolver software de modo rápido, correto e econômico permanece um objetivo utópico, e isto acontece em uma época na qual a demanda por software novo e poderoso está crescendo.

Os desenvolvedores de software estão descobrindo que usar um projeto e método de implementação modulares e orientados a objetos pode fazer com que os grupos de desenvolvimento se tornem 10 a 100 vezes mais produtivos do que seria possível com técnicas convencionais de programação.

Muitas linguagens orientadas a objetos foram desenvolvidas. A opinião geral é de que o C++ se tornará a linguagem dominante para a implementação de sistemas a partir de meados a final dos anos 90.

Muitas pessoas acreditam que a melhor estratégia educacional hoje é entender perfeitamente o C e depois estudar o C++.

1.15 Resumo

- É o software (i.e., as instruções que você escreve para ordenar ao computador a realização de ações e a tomada de decisões) que controla os computadores (chamados freqüentemente de hardware).
- ANSI C é a versão da linguagem de programação C padronizada em 1989 tanto nos Estados Unidos, através do American National Standards Institute (ANSI), como em todo o mundo, através da International Standards Organization (ISO).
- Os computadores que podem ter ocupado salas enormes e custado milhões de dólares há 25 anos podem agora estar contidos na superfície de chips de silício menores do que uma unha e que talvez custem alguns dólares cada um.
- Aproximadamente 150 milhões de computadores de uso geral estão em atividade em todo o mundo, ajudando as pessoas nos negócios, indústria, governo e em suas vidas pessoais. Este número pode dobrar facilmente em alguns anos.
- Um computador é um dispositivo capaz de realizar cálculos e tomar decisões lógicas com uma rapidez milhões, ou mesmo bilhões, de vezes maior do que os seres humanos.
- Os computadores processam dados sob o controle de programas computacionais.
- Os vários dispositivos (como teclado, tela, discos, memória e unidades de processamento) que constituem um sistema computacional são chamados de hardware.
- Os programas executados em um computador são chamados de software.
- A unidade de entrada é a seção de "recepção" do computador. Atualmente a maioria das informações é fornecida aos computadores através de teclados como os de máquinas de escrever.
- A unidade de saída é a seção de "expedição" do computador. Atualmente, a maioria das informações é fornecida pelos computadores através de exibição na tela ou impressão em papel.
- A unidade de memória é a seção de "armazenamento" do computador e é chamada freqüentemente de memória, memória principal ou memória primária.
- A unidade aritmética e lógica (arithmetic and logic unit, ALU) realiza os cálculos e toma decisões.
- A unidade central de processamento, UCP (central processing unit) é a responsável pela coordenação do computador e pela supervisão do funcionamento de outras seções.
- Normalmente, os programas ou dados que não estiverem sendo usados ativamente por outras unidades são colocados em dispositivos de armazenamento secundário (como discos) até que sejam novamente necessários.
- No processamento de lotes de usuário único (single-user batch processing), o computador executa um programa simples de cada vez enquanto processa os dados em grupos ou lotes (batches).
- A multiprogramação envolve a realização "simultânea" de várias tarefas no computador — este compartilha seus recursos entre as tarefas.
- Timesharing (tempo compartilhado) é um caso especial de multiprogramação na qual os usuários têm acesso ao computador por intermédio de terminais. Parece que os usuários estão executando programas simultaneamente.
- Com a computação distribuída, o poder computacional de uma organização é distribuído através de uma rede para os locais (sites) nos quais o trabalho real da organização é realizado.
- Os servidores de arquivo armazenam programas e dados que podem ser compartilhados por computadores clientes distribuídos ao longo da rede, daí o termo computação cliente/servidor.

- Qualquer computador pode entender diretamente sua própria linguagem de máquina.
- Geralmente, as linguagens de máquina consistem em strings de números (reduzidos em última análise a ls e Os) que mandam o computador realizar suas operações mais elementares, uma por vez. As linguagens de máquina dependem do equipamento.
- Abreviações semelhantes ao idioma inglês formam a base das linguagens assembly. Os assemblers (montadores) traduzem os programas em linguagem assembly para linguagem de máquina.
- Os compiladores traduzem os programas em linguagem de alto nível para linguagem de máquina. As linguagens de alto nível contêm palavras em inglês e notações matemáticas convencionais.
- O C é conhecido como a linguagem de desenvolvimento do sistema operacional UNIX.
- É possível escrever programas em C que sejam portáveis para a maioria dos computadores.
- O padrão ANSI C foi aprovado em 1989.
- O FORTRAN (FORmula TRANslator) é usado em aplicações matemáticas.
- O COBOL (COmmon Business Oriented Language) é usado principalmente em aplicações comerciais que exijam manipulação precisa e eficiente de grandes volumes de dados.
- Programação estruturada é um método disciplinado de escrever programas que sejam claros, visivelmente corretos e fáceis de serem modificados.
- O Pascal destinava-se ao ensino de programação estruturada em ambientes acadêmicos.
- A Ada foi desenvolvida sob o patrocínio do Departamento de Defesa dos Estados Unidos (United States" Department of Defense, DOD) usando o Pascal como base.
- A *multitarefa (multitasking)* da linguagem Ada permite aos programadores especificarem atividades paralelas.
- Todos os sistemas em C consistem em três partes: o ambiente, a linguagem e as bibliotecas padrão. As funções da biblioteca não são parte da linguagem C em si; elas realizam operações como entrada/saída de dados e cálculos matemáticos.
- Para serem executados, os programas em C passam geralmente por seis fases: edição, pré-processamento, compilação, linking (ligação), carregamento e execução.
- O programador digita um programa com um editor e faz as correções necessárias.
- Um compilador traduz um programa em C para linguagem de máquina (ou código-objeto).
- Um pré-processador obedece a diretivas que indicam normalmente que outros arquivos devem ser incluídos no arquivo a ser compilado e que símbolos especiais devem ser substituídos por texto de programa.
- Um linker liga o código-objeto ao código de funções que estejam faltando de modo a produzir uma imagem executável (com todas as partes necessárias).
- Um loader (carregador) apanha uma imagem executável do disco e a transfere para a memória.
- Um computador, sob controle de sua CPU, executa, uma a uma, as instruções de um programa.
- Determinadas funções do C (como **scanf**) recebem dados de **stdin** (o dispositivo padrão de entrada) que normalmente é atribuído ao teclado.
- Os dados são enviados a **stdout** (o dispositivo padrão de saída) que normalmente é a tela do computador.
- Há ainda um dispositivo padrão de erro chamado **stderr.** O dispositivo **stderr** (normalmente a tela) é usado para exibir mensagens de erro.

- Embora seja possível escrever programas portáteis, há muitos problemas entre as diferentes implementações do C e os diferentes computadores que podem fazer com que a portabilidade seja difícil de conseguir.
- O Concurrent C é um superconjunto do C que inclui recursos para especificar a realização de várias atividades em paralelo.
- O C++ fornece recursos para a realização de programação orientada a objetos.
- Objetos são basicamente componentes reutilizáveis de software que modelam itens do mundo real.
- A opinião geral é de que o C++ se tornará a linguagem dominante para implementação de sistemas a partir de meados a final dos anos 90.

1.16 Terminologia

Ada
ALU
ambiente
ANSI C
assembler
Biblioteca padrão do C (C Standard Library)
C
C+ +
carregador (loader)
clareza
cliente
COBOL
código-objeto
compilador
computação cliente/servidor
computação distribuída
computador
computador pessoal (personal computer; PC)
Concurrent C
CPU
dados
dependente de máquina dispositivo de entrada dispositivo de saída editor
entrada padrão (standard input, stdin)
entrada/saída (input/output, I/O)
erro padrão (standard error, stderr)
estação de trabalho (workstation)
executar um programa
extensão .c
FORTRAN
função
funcionalização
imagem executável
independente da máquina
linguagem assembly (assembly language)
linguagem de alto nível
linguagem de máquina
linguagem de programação
linguagem natural do computador
linker
memória
memória primária ou memória principal
método dos blocos de construção (building block approach)
multiprocessador
multiprogramação
multitarefa (multitasking)
objeto
Pascal
plataforma de hardware
portabilidade
pré-processador C
processamento de lotes (batch processing)
programa armazenado
programa de computador
programa tradutor
programação estruturada
programação orientada a objetos
programador de computador
refinamento top-down em etapas
reutilização de software rodar um programa
saída padrão (standard output, stdout)
servidor de arquivos
software
supercomputador
tarefa
tela
tempo compartilhado (timesharing)
terminal
unidade aritmética e lógica, UAL (arithmetic and logic unit, ALU) unidade de entrada unidade de memória
unidade de processamento central, UCP (centralprocessing unit, CPU) unidade de saída unidades lógicas UNIX

Práticas Recomendáveis de Programação

1.1 Escreva seus programas em C de uma maneira simples e objetiva. Algumas vezes isto é chamado KIS (do inglês "keep it simple" [que pode ser traduzido por "mantenha a simplicidade"]). Não "complique" a linguagem tentando soluções "estranhas".

1.2 Leia os manuais da versão do C que estiver usando. Consulte freqüentemente estes manuais para se certificar do conhecimento do rico conjunto de recursos do C e de que eles estão sendo usados corretamente.

1.3 Seu computador e compilador são bons mestres. Se você não estiver certo de como funciona um recurso do C, escreva um programa de teste que utilize esse recurso, compile e execute o programa, e veja o que acontece.

Dicas de Portabilidade

1.1 Em face de o C ser uma linguagem independente de hardware e amplamente disponível, as aplicações escritas em C podem ser executadas com pouca ou nenhuma modificação em uma grande variedade de sistemas, computacionais.

1.2 Usar as funções da biblioteca padrão do C em vez de escrever suas próprias versões similares pode melhorar a portabilidade do programa porque estas funções estão colocadas em praticamente todas as implementações do ANSI C.

1.3 Embora seja possível escrever programas portáteis, há muitos problemas entre as diferentes implementações do C e os diferentes computadores que tornam a portabilidade um objetivo difícil de atingir. Simplesmente escrever programas em C não garante a portabilidade.

Dica de Performance

1.1 Usar as funções da biblioteca padrão do C em vez de você escrever suas próprias versões similares pode melhorar o desempenho do programa porque essas funções foram desenvolvidas cuidadosamente por pessoal eficiente.

Exercícios de Revisão

1.1 Preencha as lacunas em cada uma das sentenças a seguir.

a) A companhia que criou o fenômeno da computação pessoal foi ________________.

b) O computador que validou o uso da computação pessoal no comércio e na indústria foi o ________________.

c) Os computadores processam dados sob o controle de um conjunto de instruções chamados________________.

d) As seis unidades lógicas do computador são________________, ________________, ________________,________________,________________ e ________________.

e) ________________ é um caso especial de multiprogramação na qual os usuários têm acesso ao computador através de dispositivos chamados terminais.

f). As três classes de linguagens analisadas neste capítulo são________________ , _____________ e ________________.

g) Os programas que traduzem os programas em linguagem de alto nível para linguagem de máquina são chamados ________________.

h) O C é muito conhecido como a linguagem de desenvolvimento do sistema operacional ________________.

i) Este livro apresenta a versão do C chamada ________________ C, que foi padronizada recentemente pelo American National Standards Institute.

j) A linguagem ________________ foi desenvolvida por Wirth para ensinar programação estruturada nas universidades.

k) O Departamento de Defesa dos Estados Unidos (DOD) desenvolveu a linguagem Ada com um recurso chamado ________________ que permite aos programadores especificarem muitas atividades para serem executadas em paralelo.

1.2 Preencha as lacunas em cada uma das sentenças a seguir sobre o ambiente C.

a) Os programas em C são digitados normalmente em um computador usando um programa ________________.

b) Em um sistema C, um programa ________________ é executado automaticamente antes de a fase de tradução começar.

c) Os dois tipos mais comuns de diretivas de um pré-processador são ________________ e ________________.

d) O programa ________________ combina a saída do compilador com várias funções da biblioteca para produzir uma imagem executável.

e) O programa________________ transfere a imagem executável do disco para a memória.

f) Para carregar e executar o programa compilado mais recentemente em um sistema UNIX, digite ________________.

Respostas dos Exercícios de Revisão

1.1 a) Apple
b) IBM Personal Computer
c) programas de computador
d) unidade de entrada, unidade de saída, unidade de memória, unidade aritmética e lógica (ALU), unidade de processamento central (CPU), unidade de armazenamento secundário,
e) tempo compartilhado (timesharing).
f) linguagens de máquina, linguagens assembly, linguagens de alto nível.
g) compiladores.
h) UNIX.
i) ANSI.
j) Pascal.
k) multitarefa (multitasking).

1.2 a) editor.
b) pré-processador.
c) incluindo outros arquivos a serem compilados, substituindo símbolos especiais por texto de programa.
d) linker.
e) carregador (loader).
f) **a. out.**

Exercícios

1.3 Classifique cada um dos seguintes itens como hardware ou software
a) CPU
b) compilador C
c) ALU
d) processador C
e) unidade de entrada
f) um programa editor de textos

1.4 Por que você poderia desejar escrever um programa em uma linguagem independente da máquina em vez de em uma linguagem dependente da máquina? Por que uma linguagem dependente da máquina poderia ser mais apropriada para escrever determinados tipos de programas?

1.5 Programas tradutores como os assemblers e compiladores convertem programas de uma linguagem (chamada linguagem-fonte) para outra (chamada linguagem-objeto). Determine quais das declarações a seguir são verdadeiras e quais as falsas:
a) Um compilador traduz programas em linguagem de alto nível para linguagem-objeto.
b) Um assembler traduz programas em linguagem-fonte para programas em linguagem-objeto.
c) Um compilador converte programas em linguagem-fonte para programas em linguagem-objeto.
d) Geralmente, as linguagens de alto nível são dependentes da máquina.
e) Um programa em linguagem de máquina exige tradução antes de poder ser executado no computador.

1.6 Preencha as lacunas em cada uma das frases a seguir:
a) Dispositivos dos quais os usuários têm acesso a sistemas computacionais timesharing (tempo compartilhado) são chamados geralmente ________________ .
b) Um programa de computador que converte programas em linguagem assembly para linguagem de máquina é chamado ________________ .
c) A unidade lógica do computador, que recebe informações do exterior para uso desse computador, é chamada ________________.
d) O processo de instruir o computador para resolver problemas específicos é chamado ________________.
e) Que tipo de linguagem computacional usa abreviações como as do idioma inglês para instruções em linguagem de máquina? ________________
f) Quais são as seis unidades lógicas do computador? ________________.
g) Que unidade lógica do computador envia para os vários dispositivos as informações que já foram processadas por ele, para que essas informações possam ser utilizadas em um ambiente externo ao computador? ________________
h) O nome geral de um programa que converte programas escritos em uma determinada linguagem computacional para linguagem de máquina é ________________.
i) Que unidade lógica do computador conserva as informações? ________________.
j) Que unidade lógica do computador realiza os cálculos? ________________.
k) Que unidade lógica do computador toma decisões lógicas? ________________.
1) A abreviação usada normalmente para a unidade de controle do computador é ________________.
m) O nível de linguagem computacional mais conveniente para o programador escrever

programas fácil e rapidamente é ________________.
n) A linguagem computacional destinada aos negócios mais utilizada atualmente ________________.
o) A única linguagem que um computador pode entender diretamente é ________________ chamada do computador.
p) Que unidade lógica do computador coordena as atividades de todas as outras unidades lógicas? ________________

1.7 Diga se cada uma das declarações seguintes é verdadeira ou falsa. Explique suas respostas.
a) As linguagens de máquina são geralmente dependentes do equipamento (máquina) onde são executadas.
b) O timesharing (tempo compartilhado) faz com que vários usuários realmente executem programas simultaneamente em um computador.
c) Como outras linguagens de alto nível, o C é geralmente considerado independente da máquina.

1.8 Analise o significado de cada um dos seguintes nomes do ambiente UNIX:
a) **stdin**
b) **stdout**
c) **stderr**

1.9 Qual é o principal recurso fornecido pelo Concurrent C que não está disponível no ANSI C?

1.10 Por que atualmente está sendo dedicada tanta atenção à programação orientada a objetos em geral e ao C++ em particular?

Leitura Recomendada

(An90) ANSI, ***American National Standards for Information Systems — Programming Language C (ANSI document ANSI/ISO 9899: 1990),*** **New York, NY: American National Standards Institute, 1990.**
Este é o documento que define o ANSI C. Ele está disponível para venda no American National Standards Institute, 1430 Broadway, New York, New York 10018.

(**De90) Deitei, H. M.** ***Operating Systems*** **(Second Edition), Reading, MA: Addison-Wesley Publishing Company, 1990.**
Um livro-texto para o curso tradicional de ciência da computação em sistemas operacionais. Os Capítulos 4 e 5 apresentam uma ampla análise de programação simultânea (concorrente).

(Ge89) Gehani, N.,eW. D. Roome, ***The Concurrent C Programming Language,*** **Summit, NJ: Silicon Press, 1989.**
Este é o livro que define o Concurrent C — um superconjunto da linguagem C que permite aos programadores especificarem a execução paralela de várias atividades. Inclui também um resumo do Concurrent C + + .

(Ja89) Jaeschke, R., ***Portability and the C Language,*** **Indianapolis, IN: Hayden Books, 1989.**
Este livro analisa a criação de programas portáteis em C. Jaeschke trabalhou nos comitês dos padrões ANSI e ISO.

(Ke88) Kernighan, B. W. e D. M. Ritchie, ***The C Programming Language*** **(Second Edition), Englewood Cliffs, NJ: Prentice Hall, 1988.**
Este livro é o clássico no assunto. Ele é amplamente usado em cursos e seminários de C para programadores experientes e inclui um excelente manual de referência. Ritchie é o autor da linguagem C e um dos criadores do sistema operacional UNIX.

(P192) Plauger, P.J., ***The Standard C Library,*** **Englewood Cliffs, NJ: Prentice Hall, 1992.**
Define e demonstra o uso de funções da biblioteca padrão do C. Plauger trabalhou como chefe do subcomitê da biblioteca, no comitê que desenvolveu o padrão ANSI C. e trabalha como coordenador do comitê ISO no que diz respeito ao C.

(Ra90) Rabinowitz, H.,eC. Schaap, ***Portable C,*** **Englewood Cliffs, NJ: Prentice Hall, 1990.**
Este livro foi desenvolvido para um curso sobre portabilidade realizado na AT&T Bell Laboratories. Rabinowitz está no Artificial Intelligence Laboratory da NYNEX Corporation, e Schaap é um diretor da Delft Consulting Corporation.

(Ri78) Ritchie, D. M.; S. C. Johnson; M. E. Lesk; e B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language", ***The Bell System Technical Journal,*** **Vol. 57, No. 6, Part 2, July-August 1978, pp. 1991-2019.**
Este é um dos artigos clássicos que apresentam a linguagem C. Ele foi publicado em uma edição especial do *Bell System Technical Journal* dedicado ao "UNIX Time-Sharing System".

(Ri84) Ritchie, D. M., "The UNIX System: The Evolution of the UNIX Time-Sharing System", ***AT&T Bell Laboratories Technical Journal,*** **Vol. 63, No. 8, Part *2,* October 1984, pp. 1577-1593.**
Um artigo clássico sobre o sistema operacional UNIX. Este artigo foi publicado em uma edição especial do *Bell System Technical Journal* inteiramente dedicado ao "The UNIX System".

(Ro84) Rosler, L., "The UNIX System: The Evolution of C — Past and ***Future',*** ***AT&T Bell Laboratories Technical Journal,*** **Vol. 63, No. 8. Part 2, October 1984, pp. 1685-1699.**
Um excelente artigo para vir após o (Ri78) para o leitor interessado em conhecer a história do C e as raízes dos esforços para padronização do ANSI C. Foi publicado em uma edição especial do *Bell System Technical Journal* inteiramente dedicado ao "The UNIX System".

(St84) Stroustrup, B., "The UNIX System: Data Abstraction in C", ***AT&T Bell Laboratories Technical Journal,*** **Vol. 63, No. 8, Part 2, October 1984, pp. 1701-1732.**
O artigo clássico que apresenta o C++ . Foi publicado em uma edição especial do *Bell System Technical Journal* inteiramente dedicado ao "The UNIX System".

(St91) Stroustrup, B., ***The C+ +Programming Language*** **(Second Edition), Reading, MA: Addison-Wesley Series in Computer Science, 1991.**
Este livro é a referência que define o C+ +, um superconjunto do C que inclui vários melhoramentos em relação ao C, especialmente recursos para programação orientada a objetos. Stroustrup desenvolveu o C+ + na AT&T Bell Laboratories.

Tondo, C. L., e S. E. Gimpel, ***The C Answer Book,*** **Englewood Cliffs, NJ: Prentice Hall, 1989.**
Este livro ímpar fornece respostas aos exercícios em Kernighan e Ritchie (Ke88). Os autores demonstraram exemplos de estilos de programação e demonstraram criatividade em seus métodos de resolução dos problemas e decisões de projeto. Tondo está na IBM Corporation e na Nova University em Ft. Lauderdale, Flórida. Gimpel é um consultor.

2

Introdução à Programação em C

Objetivos

- Ficar em condições de escrever programas computacionais simples em C.
- Ficar em condições de usar instruções simples de entrada e saída.
- Familiarizar-se com os tipos fundamentais de dados.
- Entender os conceitos sobre a memória do computador.
- Ficar em condições de usar os operadores aritméticos.
- Entender a precedência de operadores aritméticos.
- Ficar em condições de escrever instruções simples para tomada de decisões.

O que há em um nome? Aquilo que chamamos de rosa / Com outro nome teria o mesmo doce aroma.
William Shakespeare
Romeu e Julieta

Eu fiz apenas o curso regular ... os diferentes ramos da aritmética —Ambição, Desordem, Deturpação e Escárnio.
Lewis Carroll

Os precedentes estabelecidos deliberadamente por homens sábios merecem muito respeito.
Henry Clay

Sumário

2.1 Introdução

A linguagem C facilita o emprego de um método disciplinado e estruturado para o projeto de programas computacionais. Neste capítulo apresentamos a programação em C e mostramos vários exemplos que ilustram muitos recursos importantes da linguagem. As instruções de cada exemplo são analisadas meticulosamente, uma a uma. Nos Capítulos 3 e 4 apresentamos uma introdução à *programação estruturada* em C. Usamos então o método estruturado ao longo de todo o restante do texto.

2.2 Um Programa Simples em C: Imprimir uma Linha de Texto

O C usa algumas notações que podem parecer estranhas às pessoas que não programaram computadores. Começamos examinando um programa simples em C. Nosso primeiro exemplo imprime uma linha de texto. O programa e a tela de saída são mostrados na Fig. 2.1.

Apesar de este programa ser muito simples, ele ilustra muitos aspectos importantes da linguagem C. Agora vamos examinar detalhadamente cada linha do programa. A linha

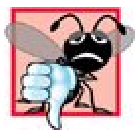
/* Primeiro programa em C */

começa com /* e termina com */ indicando que é um *comentário.* Os programadores inserem comentários para *documentar* os programas e melhorar sua legibilidade. Os comentários não fazem com que o computador realize qualquer ação quando o programa é executado. Os comentários são ignorados pelo compilador C e não fazem com que seja gerado código-objeto algum. O comentário **Primeiro programa em C** descreve simplesmente o objetivo do programa. Os comentários também servem de auxílio para outras pessoas lerem e entenderem seu programa, mas muitos comentários podem tornar um programa difícil de ler.

```
/* Primeiro programa em C */
main( )
{
printf  ("Bem-vindo ao C!\n");
}
```

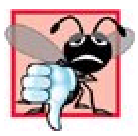
```
Bem-vindo ao C!
```

Fig. 2.1 Um programa de impressão de texto.



Erro comun de programação 2.1

Esquecer de encerrar um comentário com ***/.**



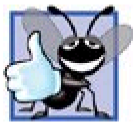
Erro comun de programação 2.2

*Começar um comentário com os caracteres */ ou terminar com /**

A linha

main( )

é uma parte de todos os programas em C. Os parênteses após a palavra **main** indicam que **main** é um bloco de construção do programa chamado *função.* Os programas em C contêm uma ou mais funções, e uma delas deve ser **main.** Todos os programas em C começam a ser executados pela função **main.**



Boas práticas de programação 2.1

Todas as funções devem ser precedidas por um comentário descrevendo seu objetivo.

A *chave esquerda, {,* deve começar o *corpo* (ou o *texto* propriamente dito) de todas as funções. Uma *chave direita* equivalente deve terminar cada função. Este par de chaves e a parte do programa entre elas também é chamado um *bloco.* O bloco é uma unidade importante dos programas em C.

A linha

printf("Bem-vindo ao C!\n");

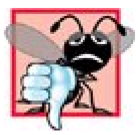
manda o computador realizar uma *ação,* especificamente imprimir na tela a *string* de caracteres limitada pelas aspas. Algumas vezes, uma string é chamada uma *string de caracteres,* uma *mensagem* ou um *valor literal.* A linha inteira, incluindo **printf,** seus *argumentos* dentro dos parênteses e o *ponto-e-vírgula* (;), é chamada uma *instrução.* Todas as instruções devem terminar com um ponto-e-vírgula (também conhecido como *marca de fim de instrução).* Quando a instrução **printf** anterior é executada, a mensagem **Bem-vindo ao C**! é impressa na tela. Normalmente os caracteres são impressos exatamente como aparecem entre as aspas duplas na instrução **printf.** Observe que os caracteres **\n** não são impressos na tela. A barra invertida (ou backslash, \) é chamada *caractere de escape.* Ele indica que **printf** deve fazer algo diferente do normal. Ao encontrar a barra invertida, **printf** verifica o próximo caractere e o combina com a barra invertida para formar uma *seqüência de escape.* A seqüência de escape **\n** significa *nova linha* e faz com que o cursor se posicione no início da nova linha na tela. Algumas outras seqüências de escape comuns estão listadas na Fig. 2.2. A função **printf** é uma das muitas funções fornecidas na *C Standard Library (Biblioteca Padrão do C,* listada no Apêndice A).

Sequencia de escape	Descrição
\n	Nova linha. Posiciona o cursor no inicio da nova linha.
\t	Tabulação horizontal. Move o cursor para a próxima marca parada de tabulação
\r	Carriage return (**CR**). Posiciona o cursor no inicio da linha atual; não avança para a próxima linha.
\a	Alerta. Faz soar a campainha (Bell), do sistema.
\\	Barra invertida (backslash). Imprime um caractere de barra invertida em uma instrução **printf**.
\”	Aspas duplas. Imprime um caractere de aspas duplas em uma instrução **printf**.

Fig 2.2 Algumas seqüências comuns de escape

As duas últimas seqüências de escape podem parecer estranhas. Como a barra invertida tem um significado especial para **printf,** i.e., **printf** a reconhece como um caractere de escape em vez de um caractere a ser impresso, usamos duas barras invertidas (\ \) para indicar que uma única barra invertida deve ser impressa. Imprimir aspas duplas também constitui um problema para **printf** porque esta instrução supõe normalmente que as aspas duplas indicam o limite de uma string e que as aspas duplas em si não devem ser realmente impressas. Usando a seqüência de escape \" dizemos a **printf** para imprimir aspas duplas.

A *chave direita, }*, indica que o fim de **main** foi alcançado.

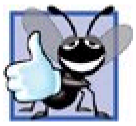


Erro comun de programação 2.3

Em um programa, digitar como print o nome da função de saída printf.

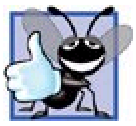
Dissemos que **printf** faz com que o computador realize uma *ação.* Durante a execução de qualquer programa, são realizadas várias ações e o programa toma *decisões.* No final deste capítulo, analisaremos a tomada de decisões. No Capítulo 3, explicaremos mais este modelo *ação/decisão* de programação.

E importante observar que funções da biblioteca padrão como **printf** e **scanf** não fazem parte da linguagem de programação C. Dessa forma, o compilador não pode encontrar um erro de digitação em **printf** e **scanf,** por exemplo. Quando o compilador compila uma instrução **printf,** ele simplesmente abre espaço no programa objeto para uma "chamada" à função da biblioteca. Mas o compilador não sabe onde as funções da biblioteca se encontram. O linker sabe. Assim, ao ser executado, o linker localiza as funções da biblioteca e insere as chamadas adequadas a elas no programa objeto. Agora o programa objeto está "completo" e pronto para ser executado. Na realidade, o programa linkeditado é chamado freqüentemente de um *executável.* Se o nome da função estiver errado, é o linker que localizará o erro, porque ele não será capaz de encontrar nas bibliotecas qualquer função conhecida que seja equivalente ao nome existente no programa cm C.



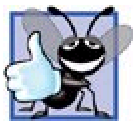
Boa prática de programação 2.2

O último caractere impresso por uma função que realiza qualquer impressão deve ser o de nova linha (\n). Isto assegura que a função deixará o cursor da tela posicionado no início de uma nova linha. Procedimentos desta natureza estimulam a reutilização do software — um objetivo principal em ambientes de desenvolvimento de software.



Boa prática de programação 2.3

Faça o recuo de um nível (três espaços) em todo o texto (corpo) de cada função dentro das chaves que a definem. Isto ressalta a estrutura funcional dos programas e ajuda a torná-los mais fáceis de ler.



Boa prática de programação 2.4

Determine uma convenção para o tamanho de recuo preferido e então aplique-a uniformemente. A tecla de tabulação (tab) pode ser usada para criar recuos, mas as paradas de tabulação podem variar. Recomendamos usar paradas de tabulação de 1/4 da polegada (aproximadamente 6 mm) ou recuar três espaços para cada nível de recuo.

A função **printf** pode imprimir **Bem-vindo ao** C! de várias maneiras. Por exemplo, o programa da Fig. 2.3 produz a mesma saída do programa da Fig. 2.**1**. Isto acontece porque cada **printf** reinicia a impressão onde o **printf** anterior termina de imprimir. O primeiro **printf** imprime **Bem-vindo** seguido de um espaço, e o segundo **printf** começa a imprimir imediatamente após o espaço.
Um **printf** pode imprimir várias linhas usando caracteres de nova linha, como mostra a Fig. 2.4. Cada vez que a seqüência de escape **\n** (nova linha) é encontrada, **printf** vai para o início da linha seguinte.

```
/* Imprimindo em uma linha com duas instruções printf */
main(  )
{
printf  /"Bem-vindo");
printf  ("ao C!\n");
}
```

```
Bem-vindo ao C !
```

Fig. 2.3 Imprimindo uma linha com instruções **printf** separadas.

```
/* Imprimindo varias linhas com um único printf */
main() {
printf  ("Bem-vindo\nao\nC!\n")
}
/* Imprimindo varias linhas com um único printf */
```

Bem-vindo
ao
C!

Fig. 2.4 Imprimindo várias linhas com uma única instrução **printf.**

2.3 Outro Programa Simples em C: Somar Dois Números Inteiros

Nosso próximo programa usa a função **scanf** da biblioteca padrão para obter dois números inteiros digitados pelo usuário, calcular a soma desses valores e imprimir o resultado usando **printf.** O programa e sua saída são mostrados na Fig. 2.5.

O comentário /*** Programa de soma ***/ indica o objetivo do programa. A linha

#include <stdio.h>

é uma diretiva para o *pré-processador C.* As linhas que começam com # são processadas pelo pré-processador antes de o programa ser compilado. Esta linha em particular diz ao pré-processador para incluir o conteúdo do *arquivo de cabeçalho de entrada/saída padrão {standard input/output headerfile,* **stdio.h).** Esse arquivo de cabeçalho (header file, chamado às vezes de arquivo de header) contém informações e instruções usadas pelo compilador ao compilar funções de entrada/saída da biblioteca padrão como **printf.** O arquivo de cabeçalho também contém informações que ajudam o compilador a determinar se as chamadas às funções da biblioteca foram escritas corretamente. Explicaremos mais detalhadamente no Capítulo 5 o conteúdo dos arquivos de cabeçalho.



Boa prática de programação 2.5

Embora a inclusão de <stdio.h> seja opcional, ela deve ser feita em todos os programas em C que usam funções de entrada/saída da biblioteca padrão. Isto ajuda o compiladora localizar os erros na fase de compilação de seu programa em vez de na fase de execução (quando normalmente os erros são mais difíceis de corrigir).

```
/* Programa de soma */
#include <stdio.h>
main(  ) {
int inteirol,  inteiro2,  soma;                    /* declaração */
printf("Entre com o primeiro inteiro\n");          /* prompt */
scanf("%d",  &inteirol);                           /* le um inteiro */
printf("Entre com o segundo inteiro\n");           /* prompt */
scanf("%d",  &inteiro2);                           /* le um inteiro */
soma = inteirol + inteiro2;                        /* atribui soma */
printf("A soma e %d/n",  soma);                    /* imprime soma */
return 0;                    /* indica que o programa foi bem-sucedido */
}
```

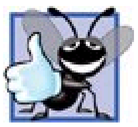
Entre com o primeiro inteiro 45
Entre com o segundo inteiro 72
A soma e 117

Fig. 2.5 Um programa de soma.

Como mencionamos anteriormente, a execução de todos os programas começa com **main.** A chave esquerda { marca o início do corpo de **main** e a chave direita correspondente marca o fim de **main.** A linha

int inteirol, inteiro2, soma;

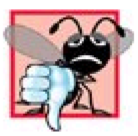
é uma *declaração.* As expressões **inteirol, inteiro2** e **soma** são os nomes das *variáveis.* Uma variável é uma posição na memória onde um valor pode ser armazenado para ser utilizado por um programa. Esta declaração especifica que **inteirol, inteiro2** e **soma** são do tipo **int** o que significa que essas variáveis conterão valores *inteiros,* i.e., números inteiros como 7, —11, 0, 31914 e similares. Todas as variáveis devem ser declaradas com um nome e um tipo de dado imediatamente após a chave esquerda que inicia o corpo de **main** antes que possam ser usadas em um programa. Em C, há outros tipos de dados além de **int.** Muitas variáveis do mesmo tipo podem estar presentes em uma declaração. Poderíamos ter escrito três declarações, uma para cada variável, mas a declaração anterior é mais concisa.



Boa prática de programação 2.6

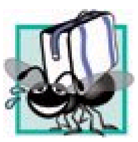
Coloque um espaço depois de cada vírgula para tornar o programa mais legível.

Um nome de variável em C é qualquer *identificador* válido. Um identificador é uma série de caracteres que consistem em letras, dígitos e sublinhados (_) que não começa com um dígito. Um identificador pode ter qualquer comprimento, mas somente os 31 primeiros caracteres serão reconhecidos pelos compiladores C, de acordo com o padrão ANSI C. O C faz distinção entre letras maiúsculas e minúsculas *(sensível a caixa alta/baixa* ou *case sensitive)* — como as letras maiúsculas e minúsculas são diferentes em C, **a1** e **A1** são identificadores diferentes.



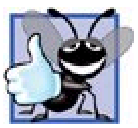
Erro comun de programação 2.4

*Usar uma letra maiúscula onde devia ser usada uma letra minúscula (por exemplo, digitar **Main** em vez de **main).***



Dicas de portabilidade 2.1

Use identificadores com 31 caracteres ou menos. Isto ajuda a assegurar a portabilidade e pode evitar alguns erros sutis de programação.



Boa prática de programação 2.7

Escolher nomes significativos para as variáveis ajuda a tornar um programa auto-explicativo, i.e., menos comentários se farão necessários.



Boa prática de programação 2.8

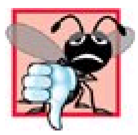
A primeira letra de um identificador usado como nome de variável simples deve ser uma letra minúscula. Mais adiante no texto atribuiremos um significado especial aos identificadores que começam com uma letra maiúscula e aos identificadores que usam todas as letras maiúsculas.



Boa prática de programação 2.9

Nomes de variáveis com mais de uma palavra podem ajudar a tornar o programa mais legível. Evite juntar palavras separadas como em totalpagamentos. Em vez disso, separe as palavras com sublinhados como em total_pagamen tos ou, se você desejar juntar as palavras, comece cada palavra depois da primeira com uma letra maiúscula como em totalPagamentos.

As declarações devem ser colocadas depois da chave esquerda e antes de *qualquer* instrução executável. Por exemplo, no programa da Fig. 2.5, inserir a declaração após o primeiro **printf** causaria um erro de sintaxe. É causado um erro de *sintaxe* quando o compilador não reconhece uma instrução. Normalmente o compilador emite uma mensagem de erro para ajudar o programador a localizar e corrigir a instrução incorreta. Os erros de sintaxe são transgressões às regras da linguagem. Eles também são chamados de *erros de compilação* ou *erros em tempo de compilação.*



Erro comun de programação 2.5

Colocar declarações de variáveis entre instruções executáveis.



Boa prática de programação 2.10

Separe as declarações das instruções executáveis em uma função por uma linha em branco, para ressaltar onde terminam as declarações e começam as instruções.

A instrução

```
printf("Entre com o primeiro inteiro\n");
```

imprime a expressão **Entre com o primeiro inteiro** na tela e se posiciona no início da próxima linha. Esta mensagem é chamada um *prompt* porque diz ao usuário para realizar uma ação específica.

A instrução

```
scanf("%d",&inteirol);
```

usa ***scanf*** para obter um valor fornecido pelo usuário. A função **scanf** recebe a entrada do dispositivo padrão, que normalmente é o teclado. Esta função tem dois argumentos **"%d"** e **&inteirol.** O primeiro argumento, a *string de controle de formato,* indica o tipo de dado que deve ser fornecido pelo usuário. O *especificador de conversão **%d*** indica que o dado deve ser um inteiro (a letra **d** significa *"decimal integer"*, o que significa em português inteiro do sistema decimal, ou seja, base 10).

Nesse contexto, o % é considerado por **scanf** (e por **printf,** como veremos) um caractere de escape (como o \) e a combinação **%d** é uma seqüência de escape (como **\n).** O segundo argumento de **scanf** começa com um e-comercial (&, ampersand, em inglês) — chamado em C de *operador de endereço* — seguido do nome da variável. O e-comercial, quando combinado com o nome da variável, diz a **scanf** o local na memória onde a variável **inteirol** está armazenada. O computador então armazena o valor de **inteirol** naquele local. Freqüentemente, o uso do e-comercial (&) causa confusão para os programadores iniciantes ou para as pessoas que programam em outras linguagens que não exigem essa notação. Por ora, basta lembrar-se de preceder cada variável em todas as instruções **scanf** com um e-comercial. Algumas exceções a essa regra são analisadas nos Capítulos 6 e 7. O significado real do uso do e-comercial se tornará claro depois de estudarmos ponteiros no Capítulo 7.

Ao executar **scanf,** o computador espera o usuário fornecer um valor para a variável **inteirol.** O usuário responde digitando um inteiro e então aperta a *tecla return* (algumas vezes chamada *tecla enter)* para enviar o número ao computador. A seguir, o computador atribui este número, ou *valor,* à variável **inteirol.** Quaisquer referências subseqüentes a **inteirol** no programa usarão esse mesmo valor. As funções **printf** e **scanf** facilitam a interação entre o usuário e o computador. Por parecer um diálogo, essa interação é chamada freqüentemente de *computação conversacional* ou *computação interativa.*

A instrução

```
printf("Entre com o segundo inteiro\n");
```

imprime a mensagem **Entre com o segundo inteiro** na tela e então posiciona o cursor no início da próxima linha. Este **printf** também faz com que o usuário realize uma ação.

A instrução

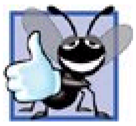
```
scanf("%d",  &inteiro2);
```

obtém o valor fornecido pelo usuário para a variável **inteiro2.** A *instrução de atribuição*

```
soma = inteirol + inteiro2;
```

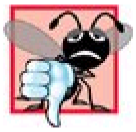
calcula o valor da soma das variáveis **inteirol** e **inteiro2**, além de atribuir o resultado à variável **soma** usando o *operador de atribuição* =. A instrução é lida como **"soma** *recebe* o valor de **inteirol + inteiro2".** A maioria dos cálculos é executada em

instruções de atribuição. Os operadores = e + são chamados *operadores binários* porque cada um deles tem *dois operandos.* No caso do operador +, os dois operandos são **inteirol** e **inteiro2.** No caso do operador =, os dois operandos são **soma** e o valor da expressão **inteirol + inteiro2 .**



Boa prática de programação 2.11

Coloque espaços em ambos os lados de um operador binário. Isto faz com que o operador seja ressaltado e torna o programa mais legível.



Erro comun de programação 2.6

O cálculo de uma instrução de atribuição deve estar no lado direito do operador =. É um erro de sintaxe colocar o cálculo no lado esquerdo de um operador de atribuição.

A instrução

```
printf("A soma e %d\n",  soma);
```

usa a função **printf** para imprimir na tela a expressão **A soma e** seguida do valor numérico de **soma.** Este **printf** tem dois argumentos, **"A soma e %d\n"** e **soma.** O primeiro argumento é a string de controle de formato. Ela contém alguns caracteres literais que devem ser exibidos e o especificador de conversão ***%ã*** indicando que um inteiro será impresso. O segundo argumento especifica o valor a ser impresso. Observe que o especificador de conversão para um inteiro é o mesmo tanto em **printf** como em **scanf.** Este é o caso da maioria dos tipos de dados em C.

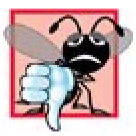
Os cálculos também podem ser realizados dentro de instruções **printf.** Poderíamos ter combinado as duas instruções anteriores na instrução

```
printf("A soma e %d\n",  inteirol + inteiro2);
```

A instrução

```
return 0;
```

passa o valor **0** de volta para o ambiente do sistema operacional no qual o programa está sendo executado. Isto indica para o sistema operacional que o programa foi executado satisfatoriamente. Para obter informações sobre como emitir um relatório com alguma espécie de falha de execução do programa, veja os manuais específicos de seu ambiente de sistema operacional. A chave direita. }, indica que o fim da função **main** foi alcançado.



Erro comun de programação 2.7

Esquecer-se de uma ou de ambas as aspas duplas em torno de uma string de controle de formato de ***printf*** *ou* ***scanf.***

Erro comun de programação 2.8

*Em uma especificação de conversão, esquecer-se do % na string de controle de formato de **printf** ou **scanf***

Erro comun de programação 2.9

*Colocar uma seqüência de escape como **n** fora da string de controle de formato de **printf** ou **scanf.***

Erro comun de programação 2.10

*Esquecer-se de incluir em uma instrução **printf** que contém especificadores de conversão as expressões cujos valores devem ser impressos.*

Erro comun de programação 2.11

*Não fornecer um especificador de conversão para uma instrução **printf,** quando tal é exigido para imprimir uma expressão.*

Erro comun de programação 2.12

Colocar, dentro de uma string de controle de formato, a vírgula que deve separar a string de controle de formato das expressões a serem impressas.

Erro comun de programação 2.13

*Esquecer-se de preceder uma variável, em uma instrução **scanf,** de um e-comercial quando essa variável deve obrigatoriamente ser precedida por ele.*

Em muitos sistemas, esse erro em tempo de execução é chamado "falha de segmentação" ou "violação de acesso". Tal erro ocorre quando o programa de um usuário tenta ter acesso a uma parte da memória do computador à qual não tem privilégios de acesso. A causa exata desse erro será explicada no Capítulo 7.

Erro comun de programação 2.14

*Preceder uma variável, incluída em uma instrução **printf,** de um e-comercial quando obrigatoriamente aquela variável não deveria ser precedida por ele.*

No Capítulo 7, estudaremos ponteiros e veremos casos nos quais desejaremos um e-comercial preceder um nome de variável para imprimir seu endereço. Entretanto, nos vários capítulos que se seguem, as instruções **printf** não devem incluir e-comerciais.

2.4 Conceitos sobre Memória

Nomes de variáveis como **inteirol, inteiro2** e **soma** correspondem realmente a *locais* na memória do computador. Todas as variáveis possuem um *nome,* um *tipo* e um *valor.* No programa de soma da Fig. 2.5, quando a instrução

scanf("%d", &inteirol);

Inteiro1	**45**

Fig. 2.6 Um local da memória mostrando o nome e o valor de uma variável.

é executada, o valor digitado pelo usuário é colocado no local da memória ao qual o nome **inteirol** foi atribuído. Suponha que o usuário digitou o número **4 5** como valor para **inteirol.** O computador colocará **45** no local **inteirol,** como mostra a Fig. 2.6. Sempre que um valor é colocado em um local da memória, o novo valor invalida o anterior naquele local. Como as informações anteriores são destruídas, o processo de levar (ler) as informações para um local da memória c chamado *leitura destrutiva (destructive read-in).*

Inteiro1	**45**
Inteiro2	**72**

Fig. 2.7 Locais de memória após a entrada de duas variáveis.

Retornando a nosso programa de soma, quando a instrução

scanf("%d", &inteiro2);

é executada, suponha que o usuário digite o valor **72.** Este valor é levado ao local **inteiro2** e a memória fica como mostra a Fig. 2.7. Observe que estas posições não são obrigatoriamente adjacentes na memória.

Depois de o programa ter obtido os valores de **inteirol** e **inteiro2,** ele os adiciona e coloca o valor da soma na variável **soma.** A instrução

soma = inteirol + inteiro2;

Inteiro1	**45**
Inteiro2	**72**
soma	**117**

Fig. 2.8 Locais da memória depois do cálculo.

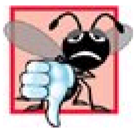
que realiza a soma também emprega leitura destrutiva. Isso ocorre quando a soma calculada de **inteirol** e **inteiro2** é colocada no local **soma** (destruindo o valor que já poderia estar ali). Depois de a **soma** ser calculada, a memória fica como mostra a Fig. 2.8. Observe que os valores de **inteirol** e **inteiro2** aparecem exatamente como antes de serem usados no cálculo da **soma.** Esses valores foram usados, mas não destruídos, quando o computador realizou o cálculo. Dessa forma, quando um valor é lido em um local da memória, o processo é chamado *leitura não-destrutiva.*

2.5 Aritmética em C

A maioria dos programas em C realiza cálculos aritméticos. Os *operadores aritméticos* do C estão resumidos na Fig. 2.9. Observe o uso de vários símbolos especiais não utilizados em álgebra. O *asterisco* (*) indica multiplicação, e o *sinal de porcentagem (%)* indica o operador *resto {modulus)* que é apresentado a seguir. Em álgebra, se quisermos multiplicar *a* por *b*, podemos simplesmente colocar lado a lado estes nomes de variáveis constituídos de uma única letra, como em *ab*. Entretanto, em C, se fizéssemos isso, **ab** seria interpretado com um único nome (ou identificador) constituído de duas letras. Portanto, o C (e outras linguagens de programação, em geral) exige que a multiplicação seja indicada explicitamente através do operador *, como em **a * b.**

Todos os operadores aritméticos são operadores binários. Por exemplo, a expressão 3 + 7 contém o operador binário + e os operandos 3 e 7 .

A *divisão inteira* leva a um resultado inteiro. Por exemplo, a expressão 7/4 leva ao resultado 1, e a expressão 17/5 leva a 3. O C possui o operador resto, %, que fornece o resto após a divisão inteira. O operador resto é um operador inteiro que só pode ser usado com operandos inteiros. A expressão ***x % y*** leva ao resto após **x** ser dividido por **y.** Dessa forma, 7 % 4 leva a 3 e l7 % 5 leva a 2. Analisaremos muitas aplicações interessantes do operador resto.



Erro comun de programação 2.15

Normalmente, uma tentativa de dividir por zero não é definida em sistemas computacionais e em geral resulta em um erro fatal, i.e., um erro que faz com que o programa seja encerrado imediatamente sem ter sucesso na realização de sua tarefa. Erros não-fatais permitem que os programas sejam executados até o final, produzindo freqüentemente resultados incorretos.

As expressões aritméticas em C devem ser escritas *no formato linear (straight-line form)* para facilitar a digitação de programas no computador. Assim, expressões como **"a** dividido por **b"** devem ser escritas como **a/b** de forma que todos os operadores e operandos apareçam em uma única linha. Em geral, a notação algébrica

***a*/b**

não é aceita pelos compiladores, embora existam alguns pacotes específicos de software que suportem notação mais natural para expressões matemáticas complexas.

Operação em C	Operador aritmético	Expresão algébrica	Expresão em C
Adição	+	f+7	f + 7
Subtração	-	p-c	p - c
Multiplicação	*	BM	b * m
Divisão	/	x/y	x / y
Resto	%	R mod s	r % s

Fig. 2.9 Operadores aritméticos do C.

Os parênteses são usados em expressões da linguagem C do mesmo modo que nas expressões algébricas. Por exemplo, para multiplicar **a** vezes a quantidade **b + c,** escrevemos:

a * (b + c)

O C calcula as expressões aritméticas em uma seqüência exata determinada pelas seguintes *regras de precedência de operadores,* que geralmente são as mesmas utilizadas em álgebra:

1. As expressões ou partes de expressões localizadas entre pares de parênteses são calculadas em primeiro lugar. Dessa forma, *os parênteses podem ser usados para impor a ordem dos cálculos segundo uma seqüência desejada pelo programador.* Diz-se que os *parênteses* estão no mais alto nível de precedência. Em casos de parênteses *aninhados* ou *embutidos,* a expressão contida no par de parênteses mais interno é calculada em primeiro lugar.

2. As operações de multiplicação, divisão e resto são calculadas a seguir. Se uma expressão possuir várias operações de multiplicação, divisão e resto, o cálculo é realizado da esquerda para a direita. Diz-se que multiplicação, divisão e resto estão no mesmo nível de precedência.

3. As operações de adição e subtração são calculadas por último. Se uma expressão possuir várias operações de adição e subtração, os cálculos são realizados da esquerda para a direita. Adição e subtração também estão no mesmo nível de precedência.

As regras de precedência de operadores são diretrizes que permitem ao C calcular expressões na ordem correta. Quando dissemos que os cálculos são realizados da esquerda para a direita, estamos nos referindo à *associatividade* de operadores. Veremos que alguns operadores se associam da esquerda para a direita. A Fig. 2.10 resume essas regras de precedência de operadores.

Agora vamos considerar várias expressões à luz das regras de precedência de operadores. Cada exemplo lista uma expressão algébrica e a expressão equivalente em C.O exemplo a seguir calcula a média aritmética de cinco termos:

Exigem-se os parênteses porque a divisão tem precedência sobre a adição. Toda a soma **(a + b + c + d + e)** deve ser dividida por **5.** Se os parênteses fossem erradamente omitidos, obteríamos **a + b + c + d + e / 5**,o que é calculado incorretamente como

Álgebra: $m = \frac{(a + b + c + d + e)}{5}$

C: m = (a + b + c + d + e) / 5;

Operador	Operação	Ordem de cálculo (precedência)
( )	Parênteses	Calculado em primeiro lugar. Se houver parênteses aninhados, a expressão dentro do par de parênteses mais interno é calculada em primeiro lugar. No caso de vários pares de parênteses “no mesmo nível” (i.e., que não estejam aninhados), eles são calculados dão esquerda para a direita.
*, / ou %	Multiplicação Divisão Resto(módulo)	Calculados em segundo lugar. No caso de vários operadores, Calculados da eles são Esquerda para direita.
+ ou -	Adição Subtração	Calculados por último. No caso de vários operadores, eles são calculados da esquerda para direita

O próximo exemplo é a equação de uma reta:

Álgebra: y = ***mx + b;***
C: **y = m * x + b;**

Não são necessários parênteses. A multiplicação é calculada em primeiro lugar por ter precedência sobre a adição.

O exemplo a seguir contém as operações resto (%), multiplicação, divisão, adição e subtração:

Álgebra: z = ***pr%q+w/x-y***
C: z = **p*r %q + w /x - Y**
(1) (2) (4) (3) (5)

Os números nos círculos abaixo da instrução indicam a ordem na qual o C calcula os operadores. As operações de multiplicação, resto e divisão são calculadas em primeiro lugar, respeitando a ordem da esquerda para a direita (i.e., sua associatividade é da esquerda para a direita) por terem nível de precedência maior do que a adição e a subtração. A adição e a subtração são calculadas a seguir. Elas também são calculadas da esquerda para a direita.

Nem todas as expressões com vários pares de parênteses contêm parênteses aninhados. A expressão

a * (b + c) + c * (d + e)

não contém parênteses aninhados. Em vez disso, diz-se que os parênteses estão no "mesmo nível". Nessa situação, o C calcula em primeiro lugar as expressões entre parênteses e seguindo a ordem da esquerda para a direita.

Para entender melhor as regras de precedência entre operadores, vamos ver como o C calcula um polinômio do segundo grau.

y =   a*x   *x + b   *x   + c;



Os números nos círculos abaixo da instrução indicam a ordem na qual o C realiza as operações. Não há operador aritmético para a exponenciação no C, por isso tivemos que representar $\mathbf{x^2}$ como **x * x.** A Biblioteca Padrão do C (C Standard Library) inclui a função **pow** (indicando "power" ou "potência") para realizar a exponenciação. Em face de algumas questões delicadas relacionadas com os tipos de dados exigidos pela função **pow,** evitaremos explicá-la detalhadamente até o Capítulo 4.

Suponha que **a = 2, b=3,c = 7 e x = 5.** A Fig. 2.11 ilustra como o polinômio de segundo grau é calculado.

2.6 Tomada de Decisões: Operadores de Igualdade e Relacionais

As instruções executáveis do C realizam *ações* (como cálculos ou entrada e saída de dados) ou tomam decisões (em breve veremos alguns exemplos disso). Podemos tomar uma decisão em um programa, por exemplo, para determinar se o grau de uma pessoa em uma prova é maior ou igual a 60 e, se for, imprimir a mensagem "Parabéns! Você passou." Esta seção apresenta uma versão simples da *estrutura de controle* ***if*** que permite que um programa tome decisões com base na veracidade ou falsidade de alguma instrução ou fato chamado *condição.* Se a condição for atendida (i.e., a condição é *verdadeira,* ou *true),* a instrução no corpo da estrutura **if** é executada. Se a condição não for atendida (i.e., a condição é *falsa,* ou *false),* a instrução do corpo da estrutura **if** não é executada. Sendo executada ou não a instrução do corpo, depois de a estrutura **if** ser concluída, a execução continua com a instrução após aquela estrutura.

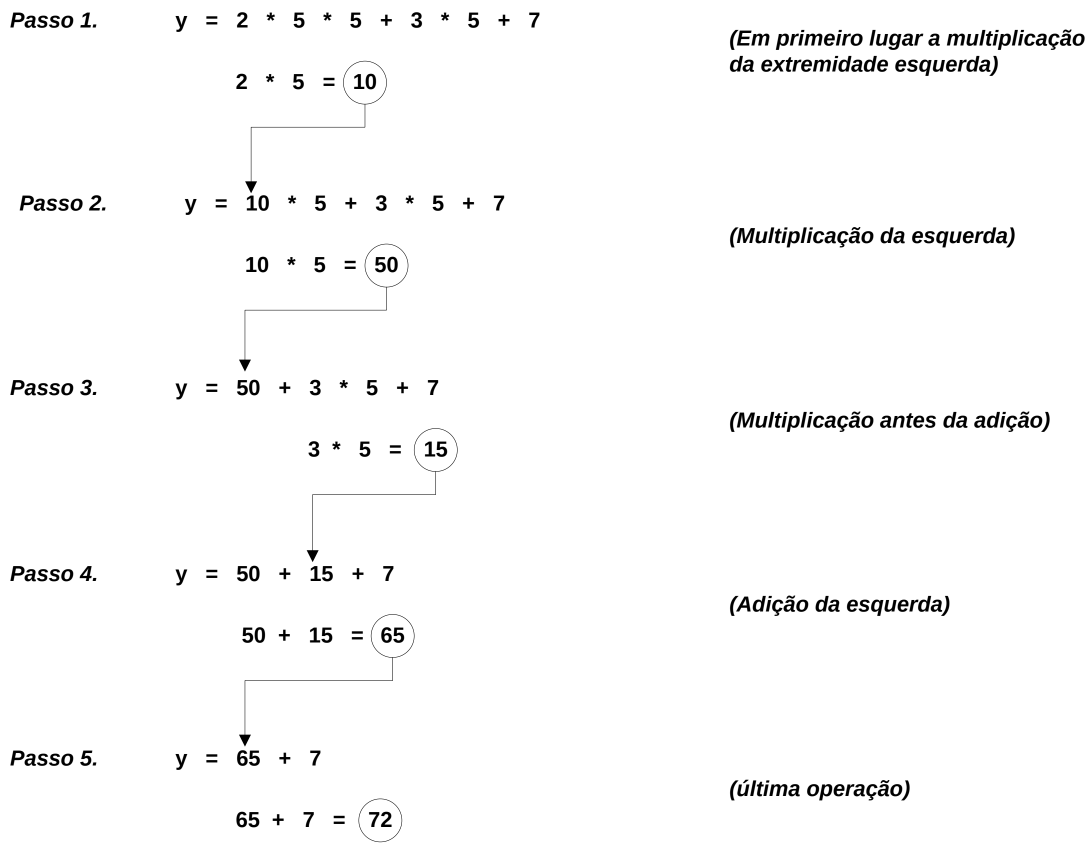
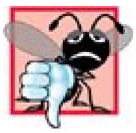


Fig. 2.11 Cálculo de um polinômio do segundo grau.

Operador algébrico padrão de igualdade ou relacional	Operador de igualdade ou relacional em C	Exemplo de condição em C	Significado da condição
Operadores de igualdade			
=	= =	x = = y	x é igual a y
≠	! =	x ! = y	x não é igual a y
Operadores relacionais			
>	>	x > y	x é maior que y
<	<	x < y	x é menor que y
≥	>=	x >= y	x é maior que ou igual a y
≤	<=	x <= y	x é maior que ou igual a y

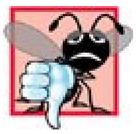
Fig. 2.12 Operadores de igualdade e relacionais,

As condições em estruturas **if** são construídas usando os *operadores de igualdade e relacionais* apresentados na Fig. 2.12. Os operadores relacionais possuem o mesmo nível de precedência e são associados da esquerda para a direita. Os operadores de igualdade possuem nível de precedência menor do que o dos operadores relacionais e também são associados da esquerda para a direita. (Nota: Em C, uma condição pode até ser qualquer expressão que gere um valor zero (falso) ou diferente de zero (verdadeiro). Veremos muitas aplicações disto ao longo do livro.)



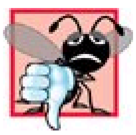
Erro comun de programação 2.16

Acontecerá um erro de sintaxe se os dois símbolos de qualquer um dos operadores ==, !=, >= e <= forem separados por espaços.



Erro comun de programação 2.17

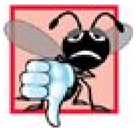
Acontecerá um erro de sintaxe se os dois símbolos em qualquer um dos operadores !=, >= e <= forem invertidos, como em =!, => e =<, respectivamente.



Erro comun de programação 2.18

Confundir o operador de igualdade == com o operador de atribuição =.

Para evitar essa confusão, o operador de igualdade deve ser lido como "é igual a" e o operador de atribuição deve ser lido como "obtém" (ou "recebe"). Como veremos em breve, confundir esses operadores pode não causar necessariamente um erro de sintaxe fácil de reconhecer, mas sim causar erros lógicos extremamente sutis.

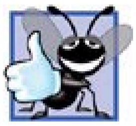


Erro comun de programação 2.19

*Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito depois de uma condição em uma estrutura **if.***

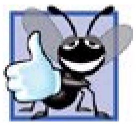
O exemplo da Fig. 2.13 usa seis instruções **if** para comparar dois números fornecidos pelo usuário. Se a condição em qualquer uma dessas instruções **if** for satisfeita, a instrução **printf** associada àquela estrutura **if** é executada. O programa e três exemplos de resultados são mostrados na figura.

Observe que o programa da Fig. 2.13 usa **scanf** para receber dois números. Cada especificador de conversão corresponde a um argumento onde um valor é armazenado. O primeiro ***%d*** converte um valor para ser armazenado na variável **numl** e o segundo **%d** converte um valor para ser armazenado na variável **num2.** Recuar o corpo de cada instrução **if** e colocar linhas em branco antes e após cada instrução **if** melhora a legibilidade do programa. Além disso, observe que cada instrução **if** da Fig. 2.13 tem uma única instrução em seu corpo. No Capítulo. 3 mostraremos como especificar instruções **if** com corpo composto de várias instruções.



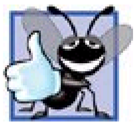
Boa prática de programação 2.12

*Recue as instruções no corpo de uma estrutura **if.***



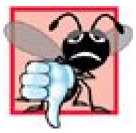
Boa prática de programação 2.13

Coloque uma linha em branco antes e após todas as estruturas de controle em um programa para melhorar sua legibilidade.



Boa prática de programação 2.14

Não deve haver mais de uma instrução por linha em um programa.



Erro comun de programação 2.20

*Colocar vírgulas (quando não são necessárias) entre os especificadores de conversão na string de controle de formato de uma instrução **scanf.***

O comentário na Fig. 2.13 ocupa duas linhas. Em programas em C, os caracteres de *espaço em branco* como tabulações, nova linha e espaços são normalmente ignorados. Assim, instruções e comentários podem ocupar várias linhas. Entretanto, não é correto dividir identificadores.

```
/* Usando instruções if,  operadores
relacionais e operadores de igualdade */
 #include <stdio.h>
main () {
int numl, num2;
printf("Entre com dois inteiros e lhe direi \n");
printf("o relacionamento que eles satisfazem:");
scanf("%d%d",  &num1,  &num2);  /* le dois inteiros */
if (num1 == num2)
        printf("%d e igual a %d\n",  num1, num2);
if  (numl  != num2)
        printf("%d nao e igual a %d\n", num1. num2);
if  (numl < num2)
        printf("%d e menor que %d\n", num1, num2);
if (numl > num2)
        printf("%d e maior que %d\n", num1, num2);
if  (numl <= num2)
        printf("%d e menor que ou igual a %d\n",  num1, num2);
if  (numl >= num2)
        printf("%d e maior que ou igual a %d\n",  num1,  num2);
return 0;  /* indica que o programa foi bem-sucedido */
}
```

```
Entre com dois inteiros e lhe direi
 o relacionamento que eles satisfazem: 3 7.
 3 nao e igual a 7
 3 e menor que 7
 3 e menor que ou igual a 7
```

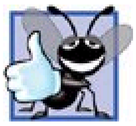
```
Entre com dois inteiros e lhe direi
o relacionamento que eles satisfazem:  22 12
22 nao e igual a 12
22 e maior que 12
22 e maior que ou igual a 12
```

```
Entre com dois inteiros e lhe direi
 o relacionamento que eles satisfazem: 7 7
 7 e igual a 7
 7 e menor que ou igual a 7 7 e maior que ou igual a 7
```

Fig. 2.13 Usando os operadores de igualdade e relacionais.

Operadores	Associatividade
( )	Esquerda para a direita
* / %	Esquerda para a direita
+ -	Esquerda para a direita
< <= > >=	Esquerda para a direita
== !=	Esquerda para direita
=	Direita para a esquerda

Fig. 2.14 Precedência e associatividade dos operadores analisados até aqui.



Boa prática de programação 2.15

Uma instrução longa pode ser dividida em várias linhas. Se uma instrução deve ocupar mais de uma linha, escolha locais de divisão que façam sentido (como após uma vírgula em uma lista separada por vírgulas). Se uma instrução for dividida em duas ou mais linhas, recue todas as linhas subseqüentes.

A tabela da Fig. 2.14 mostra a precedência dos operadores apresentados neste capítulo. Os operadores são mostrados de cima para baixo, na ordem decrescente de precedência. Observe que o sinal de igualdade também é um operador. Todos esses operadores, com exceção do operador de atribuição =, são associados da esquerda para a direita. O operador de atribuição (=) é associado da direita para a esquerda.



Boa prática de programação 2.16

Consulte a tabela de precedência dos operadores ao escrever expressões que contêm muitos operadores. Certifique-se de que os operadores da expressão são executados na ordem adequada. Se você não tiver certeza quanto à ordem de cálculo de uma expressão complexa, use parênteses para impor aquela ordem, exatamente do modo como efeito em expressões algébricas. Não se esqueça de levar em consideração que alguns operadores em C, como o operador de atribuição (=), são associados da direita para a esquerda e não da esquerda para a direita.

Algumas palavras que usamos nos programas em C deste capítulo — em particular **int, return** e **if** — são palavras-chave ou palavras reservadas da linguagem. O conjunto completo de palavras-chave em C é mostrado na Fig. 2.15. Essas palavras possuem um significado especial para o compilador C, de modo que o programador deve ter o cuidado de não as usar para identificadores, como nomes de variáveis. Neste livro, analisaremos todas essas palavras-chave.

Neste capítulo, apresentamos muitos recursos importantes da linguagem de programação C, incluindo a impressão de dados na tela, a entrada de dados pelo usuário, a realização de cálculos e a tomada de decisoes. No próximo capítulo, usamos essas técnicas como base para apresentar *programação estruturada.* O aluno ficará mais familiarizado com as técnicas de recuo (ou indentações) de instruções. Estudaremos como especificar a ordem na qual as instruções são executadas — isto é chamado de *fluxo de controle.*

Palavras-chave			
auto	break	case	Char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	uniom
unsigned	void	volatile	while

Fig. 2.15 Palavras reservadas do C.

Resumo

- Os comentários começam com / * e terminam com * /. Os programadores inserem comentários para documentar os programas e melhorar sua legibilidade. Os comentários não dão origem a nenhuma ação do computador quando o programa é executado.
- A diretiva **#include <stdio. h>** do processador diz ao compilador para incluir no programa o arquivo padrão de entrada/saída. Esse arquivo contém as informações utilizadas pelo compilador para verificar a precisão das chamadas de funções de entrada e saída, como **scanf** e **printf.**
- Os programas em C consistem em funções, e uma delas deve ser **main.** A execução de todos os programas em C começa na função **main.**
- A função **printf** pode ser usada para imprimir uma string colocada entre aspas e valores de expressões. Ao imprimir valores inteiros, o primeiro argumento da função **printf** — a string de controle do formato — contém o especificador de conversão ***%d*** e quaisquer outros caracteres que serão impressos; o segundo argumento é a expressão cujo valor será impresso. No caso de ser impresso mais de um inteiro, a string de formato de controle contém um ***%d*** para cada inteiro e os argumentos separados por vírgula, colocados depois da string de controle de formato, contêm as expressões cujos valores devem ser impressos.
- A função **scanf** obtém valores normalmente fornecidos pelo usuário através do teclado. Seu primeiro argumento é a string de formato de controle que diz ao computador que tipo de dados devem ser fornecidos pelo usuário. O especificador de conversão **%d** indica que o dado deve ser um inteiro. Cada um dos argumentos restantes corresponde a um especificador de conversão na string de controle do formato. Normalmente, cada nome de variável é precedido por um e-comercial (&), que é chamado operador de endereço em C. O e-comercial, ao ser combinado com o nome da variável, diz ao computador o local na memória onde o valor será armazenado. O computador armazena então o valor naquele local.
- Todas as variáveis de um programa em C devem ser declaradas antes que possam ser utilizadas.
- Um nome de variável em C é qualquer identificador válido. Um identificador é uma série de caracteres que consistem em letras, dígitos e sublinhados (_). Os identificadores não podem começar com um dígito. Os identificadores podem ter qualquer comprimento; no entanto, apenas os 31 primeiros caracteres são significativos, de acordo com o padrão ANSI.
- O C faz distinção entre maiúsculas e minúsculas *(case sensitivity).*
- A maioria dos cálculos é realizada em instruções de atribuição.
- Todas as variáveis armazenadas na memória do computador possuem um nome, um valor e um tipo.
- Sempre que um novo valor for colocado em um local da memória, ele substitui o valor anterior ali presente. Como as informações anteriores são destruídas, o processo de levar (ler) informações para um local da memória é chamado leitura destrutiva.
- O processo de ler um valor de um local da memória é chamado leitura não-destrutiva.
- As expressões aritméticas em C devem ser escritas em um formato linear para facilitar o fornecimento de programas para o computador.
- O C calcula as expressões aritméticas segundo uma ordem precisa determinada pelas regras de precedência de operadores e associatividade.
- A instrução **if** permite ao programador tomar uma decisão quando uma determinada condição for atendida. O formato de uma instrução **if** é

if (*condição) instrução*

Se a condição for verdadeira, a instrução no corpo do **if** é executada. Se a condição for falsa, a instrução do corpo é ignorada

- Normalmente, as condições em instruções **if** são formadas usando operadores de igualdade e relacionais. O resultado obtido ao usar esses operadores é sempre simplesmente a observação "verdadeiro" ou "falso". Observe que as condições podem ser constituídas por qualquer expressão que um valor zero (falso) ou diferente de zero (verdadeiro).

Terminologia

ação
argumento
associatividade da direita para a esquerda
associatividade da esquerda para a direita
associatividade de operadores asterisco (*)
Biblioteca Padrão do C (C Standard Library)
bloco
C
caractere de escape
caractere de escape de barra invertida (\)
caractere de escape sinal de percentagem (%)
caractere de nova linha **(\n)**
caracteres de espaço em branco
chaves { }
comentário
computação conversacional computação interativa condição
corpo de uma função
decisão
declaração
diferente de zero (verdadeiro)
distinção entre maiúsculas e minúsculas *(case sensitive)*
divisão inteira
divisão por zero
e-comercial (&, *ampersand)*
erro de compilação
erro de sintaxe
erro em tempo de compilação erro fatal
erro não-fatal
especificador de conversão especificador de conversão **%d** estrutura de controle **if**
falso
fluxo de controle formato linear função
função **printf** função **scanf** identificador
indentação
instrução de atribuição
int
inteiro
leitura destrutiva leitura não-destrutiva
literal local
local (locação) da memória
main
memória
mensagem
modelo ação/decisão
nome
nome de variável operador
operador de atribuição
operador de atribuição do sinal dc igual (=)
operador de endereço
operador multiplicação (*)
operador resto *(modulus, %)*
operadores aritméticos
operadores binários
operadores de igualdade
== "é igual a"
! = "é diferente de" operadores relacionais
> "é maior do que"
< "é menor do que"
>= "é maior do que ou igual a"
<= "é menor do que ou igual a"
operando
palavras reservadas palavras-chave palavras-chave do C
parênteses ()
parênteses aninhados (agrupados)
precedência
pré-processador C
programação estruturada
prompt
recuo
regras de precedência de operadores
seqüência de escape
stdio.h
string de caracteres
string de controle
string de controle dc formato
sublinhado (_)
tecla enter
tecla return
terminador de instrução ponto-e-vírgula (;) tipo de variável
tomada de decisão valor
valor da variável variável
verdadeiro zero (falso)

Erros Comuns de Programação

2.1 Esquecer de encerrar um comentário com */.

2.2 Começar um comentário com os caracteres */ ou terminar com /*.

2.3 Em um programa, digitar como **print** o nome da função de saída **printf.**

2.4 Usar uma letra maiúscula onde devia ser usada uma letra minúscula (por exemplo, digitar **Main** em vez de **main).**

2.5 Colocar declarações de variáveis entre instruções executáveis.

2.6 O cálculo de uma instrução de atribuição deve estar no lado direito do operador =. É um erro de sintaxe colocar o cálculo no lado esquerdo de um operador de atribuição.

2.7 Esquecer-se de uma ou ambas as aspas duplas em torno de uma string de controle de formato de **print f** ou **scanf.**

2.8 Em uma especificação de conversão, esquecer-se do % na string de controle de formato de **printf** ou **scanf.**

2.9 Colocar uma seqüência de escape como **\n** fora da string de controle de formato de **printf** ou **scanf.**

2.10 Esquecer-se de incluir em uma instrução **printf** que contém especificadores de conversão as expressões cujos valores devem ser impressos.

2.11 Não fornecer um especificador de conversão para uma instrução **printf,** quando tal é exigido para imprimir uma expressão.

2.12 Colocar, dentro de uma string de controle de formato, a vírgula que deve separar a string de controle de formato das expressões a serem impressas.

2.13 Esquecer-se de preceder uma variável, em uma instrução **scanf**, de um e-comercial quando essa variável deve obrigatoriamente ser precedida por ele.

2.14 Preceder uma variável, incluída em uma instrução **printf,** de um e-comercial quando obrigatoriamente essa variável não deveria ser precedida por ele.

2.15 Normalmente, uma tentativa de dividir por zero não é definida em sistemas computacionais e em geral resulta em um erro fatal, i.e., um erro que faz com que o programa seja encerrado imediatamente sem ter sucesso na realização de sua tarefa. Erros não-fatais permitem que os programas sejam executados até o final, produzindo freqüentemente resultados incorretos.

2.16 Acontecerá um erro de sintaxe se os dois símbolos de qualquer um dos operadores ==, ! =, >= e <= forem separados por espaços.

2.17 Acontecerá um erro de sintaxe se os dois símbolos em qualquer um dos operadores ! =, >= e <= forem invertidos, como em =!, => e =<, respectivamente.

2.18 Confundir o operador de igualdade == com o operador de atribuição =.

2.19 Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito depois de uma condição em uma estrutura **if.**

2.20 Colocar vírgulas (quando não são necessárias) entre os especificadores de conversão na string de controle de formato de uma instrução **scanf.**

Práticas Recomendáveis de Programação

2.1 Todas as funções devem ser precedidas por um comentário descrevendo seu objetivo.

2.2 O último caractere impresso por uma função que realiza qualquer impressão deve ser o de nova linha **(\n).** Isto assegura que a função deixará o cursor da tela posicionado no início de uma nova linha. Procedimentos dessa natureza estimulam a reutilização do software — um objetivo principal em ambientes de desenvolvimento de software.

2.3 Faça o recuo de um nível (três espaços) em todo o texto (corpo) de cada função dentro das chaves que a definem. Isso ressalta a estrutura funcional dos programas e ajuda a torná-los mais fáceis de ler.

2.4 Determine uma convenção para o tamanho de recuo preferido e então aplique-a uniformemente. A tecla de tabulação (tab) pode ser usada para criar recuos, mas as paradas de tabulação podem variar. Recomendamos usar paradas de tabulação de 1/4 de polegada (aproximadamente 6 mm) ou recuar três espaços para cada nível de recuo.

2.5 Embora a inclusão de **<stdio.h>** seja opcional, ela deve ser feita em todos os programas em C que usam funções de entrada/saída da biblioteca padrão. Isto ajuda o compilador a localizar os erros na fase de compilação de seu programa em vez de na fase de execução (quando normalmente os erros são mais difíceis de corrigir).

2.6 Coloque um espaço depois de cada vírgula (,) para tornar o programa mais legível.

2.7 Escolher nomes significativos para as variáveis ajuda a tornar um programa auto-explicativo, i.e., menos comentários se farão necessários.

2.8 A primeira letra de um identificador usado como nome de variável simples deve ser uma letra minúscula. Mais adiante no texto atribuiremos um significado especial aos identificadores que começam com uma letra maiúscula e aos identificadores que usam todas as letras maiúsculas.

2.9 Nomes de variáveis com mais de uma palavra podem ajudar a tornar o programa mais legível. Evite juntar palavras separadas como em **totalpagamentos.** Em vez disso, separe as palavras com sublinhados como em **total_pagamentos** ou, se você desejar juntar as palavras, comece cada palavra depois da primeira com uma letra maiúscula como em **totalPagamentos.**

2.10 Separe as declarações das instruções executáveis em uma função por uma linha em branco,

para ressaltar onde terminam as declarações e começam as instruções.

2.11 Recue as instruções no corpo de uma estrutura **if.**

2.12 Coloque uma linha em branco antes e após todas as estruturas de controle em um programa para melhora sua legibilidade.

2.13 Não deve haver mais de uma instrução por linha em um programa.

2.14 Uma instrução longa pode ser dividida em várias linhas. Se uma instrução deve ocupar mais de uma linha, escolha locais de divisão que façam sentido (como após uma vírgula em uma lista separada por vírgulas). Se uma instrução for dividida em duas ou mais linhas, recue todas as linhas subseqüentes.

2.15 Consulte a tabela de precedência dos operadores ao escrever expressões que contêm muitos operadores. Certifique-se de que os operadores da expressão são executados na ordem adequada. Se você não tiver certeza quanto à ordem de cálculo de uma expressão complexa, use parênteses para impor aquela ordem, exatamente do modo como é feito em expressões algébricas. Não se esqueça de levar em consideração que alguns operadores em C, como o operador de atribuição (=), são associados da direita para a esquerda e não da esquerda para a direita.

Dica de Portabilidade

2.1 Use identificadores com 31 caracteres ou menos. Isso ajuda a assegurar a portabilidade e pode evitar alguns erros sutis de programação.

Exercícios de Revisão

2.1 Preencha as lacunas de cada uma das frases seguintes:
a) Todos os programas em C começam sua execução com a função ________________.
b) A ________________ começa o corpo de todas as funções e a o termina.
c) Todas instruções terminam com um ________________.
d) A função ________________ da biblioteca padrão exibe informações na tela.
e) A seqüência de escape **\n** representa o caractere de ________________ que faz com que o cursor se posicione no início da próxima linha na tela.
f) A função ________________ da biblioteca padrão é usada para obter dados do teclado.
g) O especificador de conversão ________________ é usado em uma string de controle de formato de **scanf** para indicar que um inteiro será fornecido ao programa e em uma string de controle de formato de **printf** para indicar a impressão (saída) de um inteiro pelo programa.
h) Sempre que um valor novo é colocado em uma posição da memória, ele substitui o valor anterior ali presente. Esse processo é conhecido como leitura ________________ .
i) Quando um valor é lido de uma posição na memória, ele é preservado; isso é chamado leitura ________________.
j) A instrução ________________ é usada na tomada de decisões.

2.2 Diga se cada uma das afirmações seguintes é verdadeira ou falsa. Se for falsa, explique por quê.
a) Quando a função **printf** é chamada, ela sempre começa a imprimir no início de uma nova linha.
b) Os comentários fazem com que o computador imprima na tela o texto situado entre /* e */ quando o programa é executado.
c) A seqüência de escape **\n,** quando usada em uma string de controle de formato de **printf,** faz com que o cursor se posicione no início da próxima linha na tela.
d) Todas as variáveis devem ser declaradas antes de serem usadas.
e) Todas as variáveis devem receber a atribuição de um tipo ao serem declaradas.
f) O C considera idênticas as variáveis **numero** e **NuMeRo.**
g) As declarações podem aparecer em qualquer lugar do corpo de uma função.
h) Todos os argumentos após a string de controle de formato em uma função **printf** devem ser precedidos por um e-comercial (&).
i) O operador resto (%) só pode ser usado com operadores inteiros.
j) Os operadores aritméticos *, /, % e - possuem o mesmo nível de precedência.
k)Verdadeiro ou falso: Os nomes de variáveis a seguir são idênticos em todos os sistemas ANSI C:
vejaumnomesuperhiperlongol234567
vejaumnomesuperhiperlongol2**34568**

1) Verdadeiro ou falso: Um programa em C que imprime três linhas de saída deve conter três instruções **printf.**

2.3 Escreva uma instrução simples em C para realizar cada um dos pedidos que se seguem:
a) Declare do tipo **int** as variáveis **c, estaVariavel, q76354** e **numero.**
b) Peça ao usuário para fornecer um inteiro. Termine sua mensagem (prompt) com dois pontos (:) seguidos de um espaço e deixe o cursor posicionado após o espaço.
c) Leia um inteiro digitado no teclado e armazene na variável **a** o valor fornecido.
d) Se a variável **numero** não for igual a **7,** imprima **"A variável numero nao e igual a 7".**

e) Imprima a mensagem **"Este e um programa em C**" em uma linha.
f) Imprima a mensagem **"Este e um programa em C**" em duas linhas, sendo que a primeira linha termina com a palavra **um.**
g) Imprima a mensagem **"Este e um programa em C**" com cada palavra em uma linha separada.
h) Imprima a mensagem **"Este e um programa em C"** com todas as palavras separadas por tabulações.

2.4 Escreva uma instrução (ou comentário) para realizar cada um dos pedidos seguintes:
a) Crie um comentário declarando que um programa calculará o produto de três números inteiros.
b) Declare as variáveis **x, y, z** e **resultado** como sendo do tipo **int.**
c) Peça ao usuário para digitar três números inteiros.
d) Leia os três números inteiros fornecidos através do teclado e armazene-os nas variáveis **x, y** e **z.**
e) Calcule o produto dos três números inteiros contidos nas variáveis **x, y** e **z** e atribua o resultado à variável **resultado.**
f) Imprima **"O produto e"** seguido do valor da variável **resultado.**

2.5 Usando as instruções escritas para a solução do Exercício 2.4, escreva um programa completo que calcule o produto de três inteiros.

2.6 Identifique e corrija os erros de cada uma das seguintes instruções:
a) **printf("O valor e %d\n", inúmero);**
b) **scanf(*"%â%ã"*, inumerol, numero2);**
c) **if (c < 7);**
printf("C e menor do que 7\n");
d) **if (c => 7)**
printf("C e igual ou menor do que 7\n");

Respostas dos Exercícios de Revisão

2.1 a) **main.** b) chave esquerda ({), chave direita (}). c) ponto-e-vírgula, d) **printf.** e) nova linha, f) **scanf.** g)***%d.*** h) destrutiva, i) não-destrutiva. j) **if.**

2.2 a) Falso. A função **printf** sempre começa a impressão onde o cursor está posicionado, e isso pode acontecer em qualquer lugar de uma linha na tela.
b) Falso. Os comentários não fazem com seja realizada qualquer ação quando o programa é executado. Eles são usados para documentar os programas e melhorar sua legibilidade.
c) Verdadeiro.
d) Verdadeiro,
c) Verdadeiro.
f) Falso. O C faz distinção entre maiúsculas c minúsculas (ou seja, o C é *case sensitive),* portanto essas variáveis são diferentes.
g) Falso. As declarações devem aparecer depois da chave esquerda do corpo de uma função e antes de qualquer instrução executável.
h) Falso. Normalmente, os argumentos em uma função **printf** não devem ser precedidos por um e-comercial (&). Normalmente, os argumentos após a string de controle de formato em uma função **scanf** devem ser precedidos por um e-comercial. Analisaremos as exceções nos Capítulos 6 e 7.
i) Verdadeiro.
j) Falso. Os operadores *, / e % estão no mesmo nível de precedência e os operadores + e - estão em um nível inferior.
k) Falso. Alguns sistemas podem fazer distinção entre identificadores com mais de 31 caracteres. 1) Falso. Uma instrução **printf** com várias seqüências de escape **\n** pode imprimir várias linhas.

2.3 a) **int c, estaVariavel, q7 6354, numero;**
b) **printf("Entre com um numero inteiro: ");**
c) **scanf("%d", &a) ;**
d) **if (numero != 7)**
printf("O numero da variável nao e igual a 7.\n");
e) **printf("Este e um programa em C.\n");**
f) **printf("Este e um\nprograma em C.\n");**
g) **printf("Este\ne\num\nprograma\nem\nC.\n");**
h) printf("Este\te\tum\tprograma\tem\tC.\n");

2.4 a) /* **Calcule o produto de tres números inteiros */**
b) **int x, y, z, resultado;**
c) **printf("Entre com tres números inteiros: ");**
d) **scanf(*"%d%d%d"*, &x, &y, &z);**
e) **resultado = x + y + z;**
f) **printf("O produto e %d\n", resultado);**

2.5 **/* Calcule o produto de tres números inteiros */ #include <stdio.h>**
main() {
int x, y, z, resultado;
printf("Entre com tres números inteiros: "); scanf(*"%d%d%d"*, &x, &y, &z);
resultado = x + y + z;

```
printf("0 produto e %d\n",  resultado); return 0;
}
```

2.6 a) Erro: &n**úmero.** Correção: Elimine o &. Veremos mais adiante as exceções a essa regra.
b) Erro: **numero2** não tem um e-comercial. Correção: **numero2** deve ser **&numero2.** Veremos mais adiante as exceções a essa regra.
c) Erro: Ponto-e-vírgula após o parêntese direito da condição na instrução **if**. Correção: Remova o ponto-e-vírgula após o parêntese direito. Nota: O resultado desse erro é que a instrução **printf** será executada, seja verdadeira ou não a condição na instrução **if**. O ponto-e-vírgula após o parêntese direito é considerado uma instrução vazia — uma instrução que não faz nada.
d) Erro: O operador relacionai => deve ser substituído por > = .

Exercícios

2.7 Identifique e corrija os erros em cada uma das instruções a seguir (Nota: pode haver mais de um erro por instrução):

a) **scanf("d", valor);**

b) **printf("O produto de %d e *%â* e %d"\n, x, y) ;**

c) **primeiroNumero + segundoNumero = somaDosNumeros**

d) **if (numero => maior)**
maior == numero;

e) ***/ Programa para determinar o maior de tres inteiros /***

f) Scanf *("%á"*, umlnteiro) ;

g) **printf("0 resto de %d dividido por *%â* e\n", x, y, x % y);**

h) **if (x = y);**
printf(%d e igual a %d\n", x, y);

i) **print("A soma e %d\n, " x + y) ;**

j) **Printf**("O **valor fornecido e: %d\n, &valor);**

2.8 Preencha as lacunas em cada uma das expressões a seguir:

a) _________________ são usados para documentar um programa e melhorar sua legibilidade.

b) A função usada para imprimir informações na tela é _________________.

c) Uma instrução do C para a tomada de decisões é _________________.

d) Normalmente, os cálculos são realizados por instruções _________________.

e) A função _________________ fornece ao programa os valores digitados no teclado.

2.9 Escreva uma única instrução ou linha em C que realize cada um dos pedidos seguintes:

a) Imprima a mensagem **"Entre com dois números".**

b) Atribua o produto das variáveis beca variável **a.**

c) Informe que o programa realiza um exemplo de cálculo de folha de pagamento (i.e., use um texto que ajude a documentar um programa).

d) Forneça ao programa três valores inteiros digitados no teclado e coloque esses valores nas variáveis inteiras **a, b** e **c.**

2.10 Diga se cada uma das expressões a seguir é verdadeira ou falsa. Explique suas respostas, a) Os operadores em C são calculados da esquerda para a direita.

b) Todos os nomes de variáveis a seguir são válidos: **_barra_inf erior, m928134, t5, j7, suas_vendas, total_sua_conta, a, b, c, z** e **z2.**

c) A instrução **printf (" a = 5**;") ; é um exemplo típico de instrução de atribuição.

d) Uma expressão aritmética válida em C e sem parênteses é calculada da esquerda para a direita.

e) Todos os nomes de variáveis a seguir são válidos: **3g, 87,** 67h2, h22 e 2h.

2.11 Preencha as lacunas de cada uma das expressões a seguir:

a) Que operações aritméticas estão no mesmo nível de precedência que a multiplicação? .

b) Quando os parênteses são aninhados, que conjunto de parênteses de uma expressão aritmética é calculado em primeiro lugar? .

c) O local da memória do computador que pode conter valores diferentes em várias ocasiões ao longo da execução de um programa é chamado.

2.12 O que é impresso quando cada uma das instruções seguintes da linguagem C é executada? Se nada for impresso, responda "nada". Admita **x** = 2ey = **3.**

a) **printf("%d", x);**
b) **printf("%d", x + x);**
c) **printf ("x=") ;**
d) **printf("x=%d", x) ;**
e) **printf*("%ã - %â",* x** + y, y + **x);**
f) **z = x** + y;
g) **scanf(*"%ã%ã",* &x,** &y);
h) **/* printf("x** + y = ***%ã",* x** + y); */
i) printf("\n");

2.13 Quais das instruções seguintes em C, se houver alguma, contêm variáveis envolvidas com leitura destrutiva?
a) **scanf(*"%d%d%d%d%d",* &b, &c, &d, &e, &f);**
b)**p = i + j+ k + 7;**
c) **printf("Leitura destrutiva");**
d) **printf("a = 5");**

2.14 Dada a equação $y = ax^3 + 7$, qual das instruções em C a seguir, se houver alguma, são corretas para ela? a)y = **a*x*x*x + 7;**
b)y = **a*x*x* (x+7);**
c) y = **(a * x) * x * (x + 7) ;**
d) y = **(a * x) * x * x + 7;**
e) y = **a * (x * x * x) + 7;** f)y = **a*x* (x * x + 7);**

2.15 Diga a ordem de cálculo dos operadores em cada uma das instruções em C a seguir e mostre o valor de **x** depois que cada instrução for executada.
a)**x = 7 + 3***6/2-l
b)**x** = 2%2 + 2*2-2/2;
c) **x = (3 * 9 * (3 + (9*3/ (3))));**

2.16 Escreva um programa que peça ao usuário para digitar dois números, obtenha-os do usuário e imprima a soma, o produto, a diferença, o quociente e o resto da divisão dos dois números.

2.17 Escreva um programa que imprima do número 1 ao 4 na mesma linha. Escreva o programa usando os seguintes métodos:
a) Usando uma instrução **printf** sem especificadores de conversão.
b) Usando uma instrução **printf** com identificadores de conversão.
c) Usando quatro instruções **printf.**

2.18 Escreva um programa em C que peça ao usuário para fornecer dois números inteiros, obtenha-os do usuário e imprima o maior deles seguido das palavras **"e maior".** Se os números forem iguais, imprima a mensagem **"Estes números sao iguais".** Use a instrução **if** somente na forma de seleção simples que você aprendeu neste capítulo.

2.19 Escreva um programa em C que receba três números inteiros diferentes digitados no teclado e imprima a soma, a média, o produto, o menor e o maior desses números. Use a instrução **if** somente na forma ensinada neste capítulo. A tela de diálogo deve aparecer como se segue:

```
Entre com três inteiros diferentes: 13   27   14
A soma e 54
A media e 18
O produto e 4914
O menor e 13
O maior e 27
```

2.20 Escreva um programa que leia o raio de um círculo e imprima seu diâmetro, o valor de sua circunferência e sua área. Use o valor constante de 3,14159 para "pi". Faça cada um destes cálculos dentro da instrução (ou instruções) printf e use o especificador de conversão %f (Nota: Neste capítulo, analisamos apenas variáveis e constantes inteiras. No Capítulo 3 analisaremos números de ponto flutuante, i.e., valores que podem possuir pontos decimais.)

2.21 Escreva um programa que imprima um retângulo, uma elipse, uma seta e um losango como se segue:

```
********          ***         *           *
*      *        *     *      ***         * *
*      *       *       *    *****       *     *
*      *       *       *      *       *        *
*      *       *       *      *      *           *
*      *       *       *      *       *        *
*      *       *       *      *        *     *
*      *        *     *       *          * *
********          ***         *           *
```

2.22 que o seguinte código imprime?
printf("*\n**\n***\n****\n*****\n");

2.23 Escreva um programa que leia cinco números inteiros e então determine e imprima o maior e o menor inteiro do grupo. Use somente as técnicas de programação ensinadas neste capítulo.

2.24 Escreva um programa que leia um número inteiro e então determine e imprima se ele é par ou ímpar. (Dica: Use o operador resto. Um número par é múltiplo de dois. Qualquer múltiplo de dois deixa resto zero ao ser dividido por 2.)

2.25 Imprima suas iniciais em letras maiúsculas no sentido vertical, de cima para baixo, da página. Construa cada letra de sua inicial da própria letra que ela representa, do modo representado a seguir:

```
PPPPPPPPP
      P     P
      P     P
      P     P
        P  P

     JJ
   J
 J
   J
    JJJJJJJJJ

DDDDDDDDD
D                D
D                D
D                D
  D            D
    DDDDD
```

2.26 Escreva um programa que leia dois inteiros e então determine e imprima se o primeiro é múltiplo do segundo. (Dica: Use o operador resto.)

2.27 Desenhe um padrão tipo tabuleiro de xadrez utilizando instruções printf e então exiba o mesmo padrão com o menor número de instruções printf possível.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 Diga a diferença entre os termos erro fatal e erro não-fatal. Por que você poderia desejar a ocorrência de um erro fatal em vez de um erro não-fatal?

2.29 Eis um pequeno passo à frente. Neste capítulo você aprendeu a respeito de inteiros e o tipo int. O C também pode representar letras maiúsculas, letras minúsculas e uma grande variedade de símbolos especiais. O C usa internamente pequenos inteiros para representar cada caractere diferente. O conjunto de caracteres que um computador utiliza e as representações dos números inteiros correspondentes àqueles caracteres é chamado conjunto de caracteres do computador. Você pode imprimir o número inteiro equivalente à letra maiúscula **A,** por exemplo, executando a instrução

printf(*"%d"*, 'A');

Escreva um programa em C que imprima os inteiros equivalentes a algumas letras maiúsculas, letras minúsculas e símbolos especiais. No mínimo, determine os números inteiros equivalentes ao conjunto seguinte: **A BCabc 0 12** $ * + /eo caractere espaço em branco.

2.30 Escreva um programa que receba a entrada de um número de cinco dígitos, separe o número em seus dígitos componentes e os imprima separados uns dos outros por três espaços. Por exemplo, se o usuário digitar **42339,** o programa deve escrever

```
4   2   3   3   9
```

2.31 Usando apenas as técnicas aprendidas neste capítulo, escreva um programa que calcule o quadrado e o cubo dos números de 0 a 10 e use tabulações para imprimir a seguinte tabela de valores:

numero	quadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

3

Desenvolvimento da Programação Estruturada

Objetivos

- Entender as técnicas básicas para resolução de problemas.
- Ser capaz de desenvolver algoritmos através do processo de refinamento top down em etapas.
- Ser capaz de utilizar as estruturas de seleção if e if/else para definir ações.
- Ser capaz de usar a estrutura de repetição while para executar instruções repetidamente em um programa.
- Entender repetição controlada por contador e repetição controlada por sentinela.
- Entender programação estruturada.
- Ser capaz de usar os operadores de incremento, decremento e atribuição.

O segredo do sucesso é a perseverança em atingir o objetivo.
Benjamin Disraeli

Vamos todos nos mover um espaço.
Lewis Carroll

A roda completou uma volta.
William Shakespeare
O Rei Lear

Quantas maçãs caíram na cabeça de Newton até ele ter a inspiração!
Robert Frost
Comentário

Sumário

3.1 Introdução

Antes de escrever um programa para resolver uma determinada questão, é fundamental ter um completo entendimento do problema e um método cuidadosamente planejado para resolvê-lo. Os dois capítulos seguintes analisarão técnicas que facilitarão o desenvolvimento de programas computacionais estruturados. Na Seção 4.11 apresentamos um resumo da programação estruturada que junta as técnicas aqui desenvolvidas com as do Capítulo 4.

3.2 Algoritmos

A solução de qualquer problema computacional envolve a execução de uma série de ações segundo uma ordem específica. Um *procedimento (procedure)* para resolver o problema em termos de

1. as *ações* a serem executadas e
2. a *ordem* em que essas ações devem ser executadas

é chamado *algoritmo.* O exemplo a seguir demonstra a importância de especificar corretamente a ordem em que as ações devem ser executadas.

Imagine o algoritmo "acordar e trabalhar" realizado por um jovem executivo para sair da cama e ir para o trabalho:

Levantar da cama. Tirar o pijama. Tomar um banho. Vestir-se. Tomar café.
Ir de carro com colegas para o trabalho.

Essa rotina faz com que o executivo chegue ao trabalho bem preparado para tomar decisões importantes. Entretanto, suponha que as mesmas etapas fossem realizadas em uma ordem ligeiramente diferente:

Levantar da cama. Tirar o pijama. Vestir-se.
Tomar um banho.
Tomar café.
Ir de carro com colegas para o trabalho.

Nesse caso, nosso jovem executivo chegaria completamente molhado no local de trabalho. Especificar a ordem em que as instruções devem ser executadas em um programa de computador é chamado *controle do programa.* Neste e no próximo capítulo, examinaremos os recursos do C para controle do programa.

3.3 Pseudocódigo

Pseudocódigo é uma linguagem artificial e informal que ajuda os programadores a desenvolver algoritmos. O pseudocódigo que apresentamos aqui é particularmente útil para o desenvolvimento de algoritmos que serão convertidos em programas estruturados em C. O pseudocódigo é similar à linguagem do dia-a-dia; ela é conveniente e amigável, embora não seja uma linguagem real de programação.

Os programas em pseudocódigo não são na verdade executados em computadores. Em vez disso, ele simplesmente ajuda o programador a "pensar" em um programa antes de tentar escrevê-lo em uma linguagem de programação como o C. Neste capítulo, fornecemos vários exemplos de como o pseudocódigo pode ser usado eficientemente no desenvolvimento de programas em C estruturados.

O pseudocódigo consiste exclusivamente em caracteres. Desse modo, os programadores podem digitar convenientemente programas em pseudocódigo em um computador, utilizando um programa editor de textos. O computador pode exibir ou imprimir uma cópia recente do programa em pseudocódigo quando o usuário desejar. Um programa cuidadosamente elaborado em pseudocódigo pode ser convertido facilmente no programa correspondente em C. Em muitos casos, isso é feito simplesmente substituindo as instruções em pseudocódigo por suas equivalentes em C.

O pseudocódigo consiste apenas em instruções de ação — aquelas que são realizadas quando o programa for convertido do pseudocódigo para o C e executado em C. As declarações não são instruções executáveis. Elas são mensagens para o compilador. Por exemplo, a declaração

int i;

simplesmente diz ao compilador o tipo da variável **i** e dá a ordem para que ele reserve espaço na memória para esta variável. Mas essa declaração não dá origem a qualquer ação — como uma entrada de dados, saída de resultados ou cálculo — quando o programa for executado. Alguns programadores preferem listar todas as variáveis no início de um programa em pseudocódigo e ali mencionar brevemente a finalidade de cada uma delas. Mais uma vez, o pseudocódigo é uma ajuda informal para o desenvolvimento de programas.

3.4 Estruturas de Controle

Normalmente, as instruções em um programa são executadas uma após a outra, na ordem em que foram escritas. Isto é chamado *execução seqüencial.* Várias instruções em C que analisaremos em breve permitem que o programador especifique que a próxima instrução a ser executada seja diferente da próxima na seqüência. Isto é chamado *transferência de controle.* Durante os anos 60, ficou claro que o uso indiscriminado de transferências de controle era a causa da grande dificuldade que os grupos de desenvolvimento de software enfrentavam. A responsabilidade foi atribuída à *instrução* **goto** que permite ao programador especificar uma transferência de controle para um ou vários destinos possíveis em um programa. A noção da conhecida *programação estruturada* se tornou quase sinônimo de *"eliminação do* **goto".**

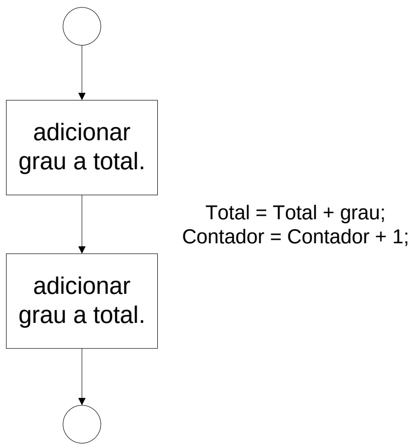


Fig. 3.1 Fluxograma de uma estrutura de seqüência em C.

A pesquisa de Bohm e Jacopini[1] demonstrou que era possível escrever programas sem qualquer instrução **goto.** O desafio da época para os programadores se tornou modificar seus estilos para adotar uma "programação sem **goto".** Somente depois de meados dos anos 70 é que os programadores em geral começaram a levar a programação estruturada a sério. Os resultados foram impressionantes, na medida em que os grupos de desenvolvimento de software apresentavam tempo reduzido para desenvolvimento, entrega mais freqüente de sistemas dentro do prazo e conclusão de projetos de software dentro dos orçamentos previstos. O segredo desse sucesso foi simplesmente que os produzidos com tais técnicas estruturadas eram mais claros, mais fáceis de depurar e modificar, e principalmente com maior probabilidade de estarem livres de erros (bug-free).

O trabalho de Bohm e Jacopini demonstrou que todos os programas podiam ser escritos em termos de apenas três *estruturas de controle,* que eram a *estrutura de seqüência (sequencial), a estrutura de seleção* e *a estrutura de repetição.* A estrutura de seqüência está essencialmente inserida no C. A menos que seja ordenado de outra forma, o computador executa automaticamente as instruções do C, uma após a outra, na ordem em que foram escritas. O segmento de *fluxograma* da Fig. 3.1 ilustra a estrutura de seqüência do C.

Um fluxograma é uma representação de todo um algoritmo ou de uma parte dele. Os fluxogramas são desenhados usando símbolos com significado especial como retângulos, losangos, elipses e pequenos círculos; esses símbolos são conectados por setas chamadas *linhas de fluxo (flowlines).*

Da mesma forma que o pseudocódigo, os fluxogramas são úteis para o desenvolvimento e a representação de algoritmos, embora o pseudocódigo seja preferido pela maioria dos programadores. Os fluxogramas mostram claramente como funciona o controle das estruturas; eles serão utilizados apenas para isso neste texto.

Considere o segmento de fluxograma para a estrutura seqüencial da Fig. 3.1. Usamos o símbolo na forma de um *retângulo,* também chamado *símbolo de ação,* para indicar qualquer tipo de ação incluindo um cálculo ou uma operação de entrada/saída. As linhas de fluxo na figura indicam a ordem na qual as ações são realizadas — em primeiro lugar, **grau** deve ser adicionado a **total** e então **1** deve ser adicionado a **contador.** A linguagem C permite que tenhamos tantas ações quanto quisermos em uma estrutura seqüencial. Como veremos em breve, em qualquer lugar em que uma única ação pode ser colocada, será possível colocar várias ações em seqüência.

Ao desenhar um fluxograma que representa um algoritmo *completo,* um símbolo em forma de *elipse* contendo a palavra "Início" (ou "Begin") é o primeiro símbolo usado no fluxograma; uma elipse contendo a palavra "Fim" ("End") é o último símbolo utilizado. Ao desenhar apenas uma parte de um algoritmo, como na Fig. 3.1, os símbolos em forma de elipse são omitidos e são utilizados símbolos no formato de *pequenos círculos,* chamados *símbolos de conexão (símbolos conectores).*

Talvez o símbolo mais importante do fluxograma seja o que está em formato de *losango,* também chamado *símbolo de decisão,* que indica que uma decisão deve ser tomada. Examinaremos o símbolo com formato de losango na próxima seção.

A linguagem C fornece três tipos de estruturas de seleção. A estrutura de seleção **i f** (Seção 3.5) realiza (seleciona) uma ação se uma condição for verdadeira ou ignora a ação se a condição for falsa. A estrutura de seleção **if/else** (Seção 3.6) realiza uma ação se uma condição for verdadeira e realiza uma ação diferente se a condição for falsa. A estrutura de seleção **switch** (analisada no Capítulo 4) realiza uma entre muitas ações diferentes dependendo do valor de uma expressão.

A estrutura **i f** é chamada *estrutura de seleção simples (única)* porque seleciona ou ignora uma única ação. A estrutura de seleção **if/else é** chamada *estrutura de seleção dupla* porque seleciona uma entre duas ações diferentes. A estrutura de seleção **switch** é chamada *estrutura de seleção múltipla* porque seleciona uma entre muitas ações diferentes.
A linguagem C fornece três tipos de estruturas de repetição, que são a estrutura **while** (Seção 3.7) e as estruturas **do/while** e **for** (ambas analisadas no Capítulo 4).

Isso é tudo. O C tem apenas sete estruturas de controle: seqüenciais, três tipos de estruturas de seleção e três tipos de estruturas de repetição. Todos os programas em C são construídos através da combinação de tantas estruturas de cada tipo quanto for adequado para o algoritmo do programa implementado. Da mesma forma que a estrutura de seqüência da Fig. 3.1, veremos que cada estrutura de controle possui dois símbolos no formato de pequenos círculos, um no ponto de entrada da estrutura de controle e outro no ponto de saída. Essas *estruturas de controle de única-entrada/única-saída* facilitam a construção de programas. As estruturas de controle podem ser associadas entre si conectando o ponto de saída de uma com o ponto de entrada da estrutura seguinte. Isso é muito parecido com o modo pelo qual uma criança empilha blocos de construção, por isso é chamado *empilhamento (superposição) de estruturas de controle.* Aprenderemos que há somente uma outra maneira pela qual as estruturas de controle podem ser conectadas — um método

chamado *aninhamento de estruturas de controle.* Assim, todos os programas em C que precisaremos criar podem ser construídos a partir de apenas sete tipos diferentes de estruturas de controle combinados de apenas duas maneiras.

3.5 A Estrutura de Seleção IF

A estrutura de seleção if é usada para se fazer uma escolha entre várias linhas de ação alternativas. Por exemplo, suponha que o grau de aprovação em um exame é 60. A instrução seguinte em pseudocódigo

Se o grau do aluno for maior que ou igual que 60 Imprimir "Aprovado "

determina se a condição "grau do aluno for maior que ou igual a 60" é verdadeira ou falsa. Se a condição for verdadeira, a palavra "Aprovado" é impressa e a instrução que vem logo após o pseudocódigo é "realizada" (lembre-se de que o pseudocódigo não é uma linguagem real de programação). Se a condição for falsa, a impressão é ignorada e a instrução que vem logo após o pseudocódigo é realizada. Observe que a segunda linha dessa estrutura de seleção está recuada (indentada). Tal recuo é opcional, mas altamente recomendado por destacar a estrutura inerente dos programas estruturados. Aplicaremos criteriosamente as convenções para os recuos (indentações) ao longo deste texto. O compilador C ignora *caracteres em branco (whitespace characters)* como os caracteres de espaço, tabulação e nova linha usados em recuos e no espaçamento vertical.



Boa prática de programação 3.1

Aplicar consistentemente as convenções para os recuos aumenta bastante a legibilidade do programa. Sugerimos valores fixos de aproximadamente 1/4 da polegada ou três espaços em branco em cada nível de recuo.

A instrução *If* do pseudocódigo anterior pode ser escrita em C como

```
if  (grau >= 60)
printf("Aprovado\n");
```

Observe que o código em C tem muita semelhança com a versão do pseudocódigo em inglês. Esta é uma das propriedades do pseudocódigo que o torna uma grande ferramenta para o desenvolvimento de programas.



Boa prática de programação 3.2

Freqüentemente, o pseudocódigo é usado para "elaborar" um programa durante o processo de projeto do programa. Depois o programa em pseudocódigo é convertido para o C.

O fluxograma da Fig. 3.2 ilustra uma estrutura **if** de seleção simples. Esse fluxograma contém o que talvez seja seu símbolo mais importante — o símbolo com formato de *losango,* também chamado *símbolo de decisão,* que indica uma decisão a ser tomada. O símbolo de decisão contém uma expressão, como uma condição, que pode ser verdadeira ou falsa. O símbolo de decisão dá origem a duas linhas de fluxo. Uma indica a direção a tomar quando a expressão no interior do símbolo for verdadeira; a outra indica a direção a tomar quando ela for falsa. Aprendemos no Capítulo 2 que as decisões podem ser tomadas com base em condições contendo operadores relacionais ou de igualdade. Na realidade, uma decisão pode ser tomada com qualquer expressão — se o cálculo da

expressão leva ao valor zero, ela é considerada falsa, e se o cálculo da expressão leva a um valor diferente de zero, ela é considerada verdadeira.

Observe que a estrutura **if** também é uma estrutura de única-entrada/única-saída. Veremos em breve que os fluxogramas para as estruturas de controle restantes também conterão (além dos símbolos com o formato de pequenos círculos e linhas de fluxo) apenas símbolos no formato de retângulos, para indicar as ações a serem realizadas, e losangos, para indicar as decisões a serem tomadas. Esse é o modelo ação/decisão de programar que estivemos destacando.

Podemos imaginar sete latas, cada uma delas contendo estruturas de controle de apenas um dos sete tipos. Essas estruturas de controle estão vazias. Nada está escrito nos retângulos e nos losangos. Assim, a tarefa do programador é montar um programa utilizando tantas estruturas de controle de cada tipo quantas o algoritmo exigir, combinando-as de apenas duas maneiras possíveis (empilhamento ou aninhamento) e então preenchendo as ações e decisões de forma apropriada para o algoritmo. Analisaremos as várias maneiras possíveis de escrever ações e decisões.

3.6 A Estrutura de Seleção If/Else

A estrutura de seleção **if** realiza uma ação indicada somente quando a condição for verdadeira; caso contrário a ação é ignorada. A estrutura de seleção **if/else** permite ao programador especificar que sejam realizadas ações diferentes conforme a condição seja verdadeira ou falsa. Por exemplo, a instrução em pseudocódigo

Se o grau do aluno for maior que ou igual a 60
Imprimir "Aprovado " senão
Imprimir "Reprovado"

imprime *Aprovado* se o grau do aluno for maior que ou igual a 60 e imprime *Reprovado* se o grau do aluno for menor que 60. Em qualquer um dos casos, depois de a impressão acontecer, a instrução que vem após o 3pseudocódigo na seqüência é realizada. Observe que o corpo de *senão (else)* também está recuado.

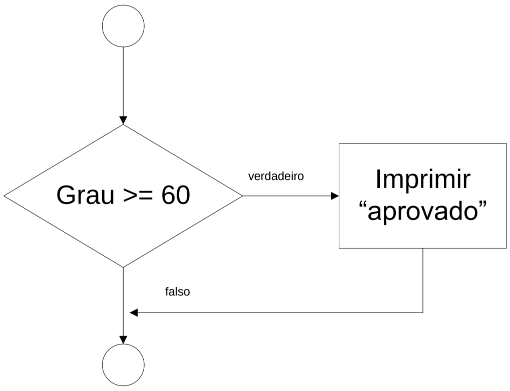


Fig. 3.2 Fluxograma de uma estrutura de seleção simples em C.



Boa prática de programação 3.3

Aplique recuos nas instruções em ambas as partes da estrutura ***if/else.***

A convenção de recuo escolhida, qualquer que seja ela, deve ser aplicada criteriosamente em todos os seus programas. É difícil ler um programa que não obedeça a convenções de espaçamento uniforme.



Boa prática de programação 3.4

Se houver vários níveis de recuos, o recuo de cada nível deve ter a mesma quantidade adicional de espaços.

A estrutura *If/else* do pseudocódigo anterior pode ser escrita em C como

```
if  (grau >= 60)
printf("Aprovado \n"); else
printf("Reprovado \n");
```

O fluxograma da Fig. 3.3 ilustra bem o fluxo de controle da estrutura **if/else.** Mais uma vez, observe que (além dos pequenos círculos e setas) os únicos símbolos no fluxograma são retângulos (para as ações) e losangos (para uma decisão). Continuamos a enfatizar esse modelo computacional de ação/decisão. Imagine novamente uma grande lata contendo tantas estruturas de seleção dupla quantas forem necessárias para construir um programa em C. Novamente a tarefa do programador é combinar essas estruturas (por empilhamento ou aninhamento) com qualquer outra estrutura de controle exigida pelo algoritmo e preencher os retângulos e losangos vazios com ações e decisões adequadas ao algoritmo que está sendo implementado.

A linguagem C fornece o *operador condicional (?:)* que está intimamente relacionado com a estrutura **if/else.** O operador condicional é o único *operador ternário* do C — ele utiliza três operandos. Os operandos e o operador condicional formam uma *expressão condicional.* O primeiro operando é uma condição, o segundo, o valor de toda a expressão condicional se a condição for verdadeira, e o terceiro, o valor de toda a expressão condicional se a condição for falsa. Por exemplo, a instrução **printf**

```
printf("%s\n",  grau >= 60 ?  "Aprovado"   : "Reprovado");
```

contém uma expressão condicional que assume o valor da string literal **"Aprovado"** se a condição **grau >= 60** for verdadeira e assume o valor " **Reprovado**" se a condição for falsa. A string de controle de formato de **printf** contém a especificação de conversão ***%s*** para imprimir uma string de caracteres. Assim, a instrução **printf** anterior realiza essencialmente o mesmo que a instrução **if/else** precedente.

Os valores em uma expressão condicional também podem ser ações a serem executadas. Por exemplo, a expressão condicional

```
grau >= 60 ? printf("Aprovado\n")   : printf("Reprovado\n");
```

é lida desta forma, "Se **grau** for maior ou igual a **60** então execute **printf (" Aprovado\n")**, caso contrário execute **printf ("Reprovado\n")**. Isso também faz o mesmo que a estrutura **if/else** anterior. Veremos que os operadores condicionais podem ser usados em algumas situações em que instruções **if/else** não podem.

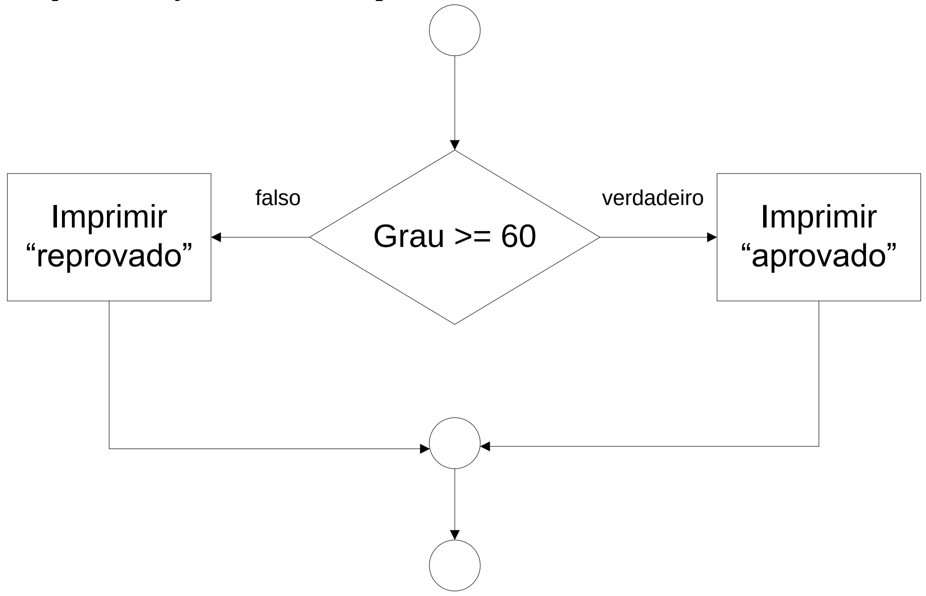


Fig. 3.3 Fluxograma de uma estrutura **if**/else de seleção dupla em C,

Estruturas ***if/else*** *aninhadas* verificam vários casos inserindo umas estruturas **if/else** em outras. Por exemplo, a instrução em pseudocódigo a seguir imprimirá **A** para graus de exame maiores que **9 0,B** para graus maiores que ou iguais a **80,C** para graus maiores que ou iguais a **70,D** para graus maiores que ou iguais a **60** e **F** para todos os outros graus.

```
Se o grau do aluno for maior que ou igual a 90
        Imprimir "A " senão
Se o grau do aluno for maior que ou igual a 80
        Imprimir "B" senão
Se o grau do aluno for maior que ou igual a 70
        Imprimir "C" senão
Se o grau do aluno for maior que ou igual a 60
        Imprimir "D" senão
        Imprimir "F"
```

Esse pseudocódigo pode ser escrito em C como

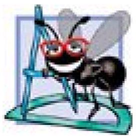
```
if  (grau >= 90)
        printf("A\n");          else
if  (grau >= 80)
        printf("B\n");          else
if  (grau >= 70)
        printf("C\n");          else
if  (grau >= 60)
        printf("D\n");          else
        printf("F\n");
```

Se a variável **grau** for maior que ou igual a 90, as quatro primeiras condições serão verdadeiras, mas somente a instrução **printf** colocada antes do primeiro teste será executada. Depois de aquele **printf** ser executado, a porção **else** do **if/else** "externo" é ignorada. Muitos programadores em C preferem escrever a estrutura **i f** anterior como

```
if  (grau >= 90)
        printf("A\n");
else if  (grau >= 80)
        printf("B\n");
else if  (grau >= 70)
        printf("C\n");
else if  (grau >= 60)
        printf("D\n");
else
        printf("F\n") ;
```

Para o compilador C, ambas as formas são equivalentes. A última forma é mais popular porque evita recuos muito grandes para a direita. Freqüentemente, tais recuos deixam um espaço muito pequeno para uma linha, obrigando a que as linhas sejam divididas e prejudicando a legibilidade do programa.

A estrutura de seleção **if** deve conter apenas uma instrução em seu corpo. Para incluir várias instruções no corpo de um **if**, coloque o conjunto de instruções entre chaves ({ e }). Um conjunto de instruções dentro de um par de chaves é chamado uma *instrução composta.*



Observação de engenharia de software 3.1

Uma instrução composta pode ser colocada em qualquer lugar de um programa no qual pode ser colocada uma instrução simples.

O exemplo a seguir inclui uma instrução composta na porção **else** de uma estrutura **if/else.**

```
if  (grau >= 60)
        printf("Aprovado.\n");
else {
        printf("Reprovado.\n");
        printf("Voce deve fazer este curso novamente.\n");
}
```

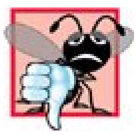
Nesse caso, se o grau for menor que **60,** o programa executa ambas as instruções **printf** no corpo de **else** e imprime

Reprovado.
Voce deve fazer este curso novamente.

Observe as chaves em torno das duas instruções na cláusula **else.** Elas são importantes. Sem elas, a instrução

```
printf("Voce deve fazer este curso novamente.\n");
```

estaria no lado de fora da porção **else do if e** seria executada independentemente de o grau ser menor que 60.



Erro comun de programação 3.1

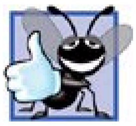
Esquecer de uma ou ambas as chaves que limitam uma instrução composta.

Os erros de sintaxe são acusados pelo compilador. Os erros lógicos produzem seus efeitos durante o tempo de execução. Um erro lógico fatal faz com que o programa falhe e termine prematuramente. Um erro lógico não-fatal permite que o programa continue a ser executado mas produz resultados incorretos.



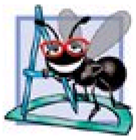
Erro comun de programação 3.2

*Colocar um ponto-e-vírgula (;) depois da condição em uma estrutura **if** leva a um erro lógico em estruturas **if** de seleção simples e a um erro de sintaxe em estruturas **if** de seleção dupla.*



Boa prática de programação 3.5

Alguns programadores preferem digitar as chaves inicial e final de instruções compostas antes de digitar cada uma das instruções incluídas nessas chaves. Isso ajuda a evitar a omissão de uma ou ambas as chaves.



Observação de engenharia de software 3.2

Da mesma forma que uma instrução composta pode ser colocada em qualquer local onde uma instrução simples pode estar, também é possível não ter instrução alguma, i.e., uma instrução vazia. A instrução vazia é representada colocando um ponto-e-vírgula (;) onde normalmente a instrução deveria estar.

Nesta seção, apresentamos a noção de instrução composta. Uma instrução composta pode conter declarações (como o corpo de **main,** por exemplo). Se isso acontecer, a instrução composta é chamada bloco. As declarações em um bloco devem ser as primeiras linhas do bloco, antes de qualquer instrução de ação. Analisaremos o uso de blocos no Capítulo 5. O leitor deve evitar o uso de blocos (além do corpo de **main,** obviamente) até lá.

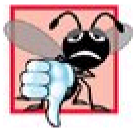
3.7 A Estrutura de Repetição While

Uma estrutura de repetição permite ao programador especificar que uma ação deve ser repetida enquanto uma determinada condição for verdadeira. A instrução em pseudocódigo

Enquanto houver mais itens em minha lista de compras Comprar o próximo item e riscá-lo de minha lista

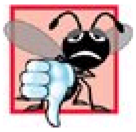
descreve uma repetição que ocorre durante a realização das compras. A condição "houver mais itens em minha lista de compras" pode ser verdadeira ou falsa. Se for verdadeira, a ação "Compre o próximo item e risque-o de minha lista" é realizada. Essa ação será realizada repetidamente enquanto a condição permanecer verdadeira. A instrução (ou instruções) contida na estrutura de repetição *while* constitui o corpo do *while.* O corpo da estrutura *while* pode ser uma instrução simples ou composta.

Posteriormente, a condição se tornará falsa (quando o último item da lista for comprado e riscado na lista). Nesta ocasião, a repetição termina, e é executada a primeira instrução do pseudocódigo colocada logo após a estrutura de repetição.



Erro comun de programação 3.3

*Fornecer no corpo de uma estrutura **while** uma ação que posteriormente não torna falsa a condição no **while.** Normalmente, tal estrutura de repetição nunca terminará — este erro é chamado "loop infinito".*



Erro comun de programação 3.4

*Digitar a palavra-chave **while** com letra **W**(maiúscula) como em **While** (lembre-se de que o C é uma linguagem que faz distinção entre letras maiúsculas e minúsculas). Todas as palavras reservadas do C como **while,** **if** e **else** contêm apenas letras minúsculas.*

Como exemplo de um **while** real, considere o trecho de programa destinado a encontrar a primeira potência de 2 maior que 1000. Suponha que a variável inteira **produto** foi inicializada com o valor 2. Quando a estrutura de repetição a seguir terminar sua execução, **produto** conterá a resposta procurada:

```
produto = 2;
while  (produto <= 1000) produto = 2 * produto;
```

O fluxograma da Fig. 3.4 ilustra bem o fluxo do controle na estrutura de repetição **while.** Mais uma vez, observe que (além dos pequenos círculos e das setas) o fluxograma contém apenas um retângulo e um losango. Imagine novamente uma grande lata de estruturas **while** vazias que podem ser empilhadas ou aninhadas com outras estruturas de controle para formar uma implementação estruturada do fluxo de controle de um algoritmo. Os retângulos e losangos vazios são então preenchidos com as ações e decisões apropriadas. O fluxograma mostra claramente a repetição. A linha de fluxo que sai do retângulo volta para a decisão que é testada novamente ao longo do loop até que seja considerada falsa.

Nessa ocasião, a estrutura **while** é abandonada e o controle passa para a próxima instrução no programa.

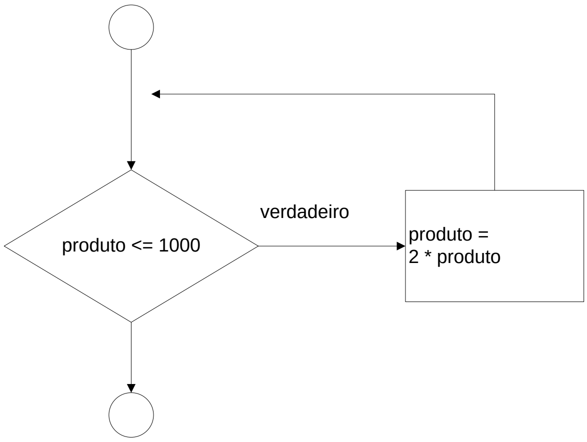


Fig. 3.4 Fluxograma da estrutura de repetição **while.**

Ao ser iniciada a estrutura **while** o valor de **produto** é 2. A variável **produto** é multiplicada repetidamente por 2, assumindo os valores 4, 8,16, 32, 64, 128, 256, 512 e 1024 sucessivamente. Quando **produto** se tornar 1024, a condição na estrutura **while, produto <= 1000,** se torna falsa. Isto encerra a repetição e o valor final de **produto** será 1024. A execução do programa continua com a instrução seguinte à estrutura **while.**

3.8 Formulando Algoritmos: Estudo de Caso 1 (Repetição Controlada por Contador)

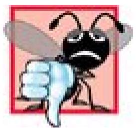
Para ilustrar como os algoritmos são desenvolvidos, resolvemos várias formas de um problema de média de notas de uma turma de alunos. Considere o seguinte enunciado de um problema:

Uma turma de dez alunos fez um teste. Os graus ,(inteiros variando de 0 a 100) do teste estão disponíveis para você. Determine a média da turma no teste.

A média da turma é igual à divisão da soma dos graus pelo número de alunos. O algoritmo para resolver esse problema em um computador deve receber cada um dos graus, realizar o cálculo da média e imprimir o resultado.

Vamos usar o pseudocódigo, listar as ações a serem executadas e especificar a ordem em que elas devem ser realizadas. Usamos a *repetição controlada por contador* para obter um grau de cada vez. Essa técnica usa uma variável chamada *contador* para especificar o número de vezes que um conjunto de instruções deve ser executado. Nesse exemplo, a repetição termina quando o contador supera 10. Nesta seção simplesmente apresentamos o algoritmo do pseudocódigo (Fig. 3.5) e programa em C correspondente (Fig. 3.6). Na próxima seção, mostramos como os algoritmos de pseudocódigo são desenvolvidos. A repetição controlada por contador é chamada normalmente *repetição definida* porque o número de repetições é conhecido antes de o loop começar a ser executado.

Observe as referências no algoritmo a um total e a um contador. Um *total* é uma variável usada para acumular uma série de valores. Um contador é uma variável usada para contar — neste caso, para contar o número de graus fornecidos. Normalmente, as variáveis usadas para armazenar totais devem ser inicializadas com o valor zero antes de serem utilizadas em um programa; caso contrário a soma incluiria os valores anteriores armazenados no local de memória daquele total. Normalmente as variáveis que servem para contadores são inicializadas com o valor zero ou um, dependendo de seu uso (apresentaremos exemplos mostrando cada um desses usos). Uma variável não-inicializada contém um *valor "lixo"* — o último valor armazenado no local da memória reservado para aquela variável.



Erro comun de programação 3.5

Se um contador ou total não for inicializado, provavelmente os resultados de seu programa estarão incorretos. Esse é um exemplo de erro lógico. Definir total com o valor zero Definir contador com o valor 1.
Enquanto o contador de graus for menor que ou igual a dez Obter próximo grau Adicionar o grau ao total Adicionar um ao contador de graus.
Definir a média da turma com o valor total dividido por dez Imprimir a média da turma.

Definir o total com valor 0
Definir contador com valor 1

Enquanto o contador de graus for menor que ou igual a dez
    Obter próximo grau
    Adicionar o grau ao total
    Adicionar um ao contador de graus

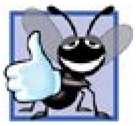
Definir a média da turma com o valor total dividido por dez
Imprimir a média da turma

Fig. 3.5 Algoritmo do pseudocódigo que usa repetição controlada por contador para resolver o problema da média da turma.

```
/*Programa de media da turma com
repetição controlada por contador */
#include <stdio.h>
main (){
int contador,  grau,  total, media;
/* fase de inicialização */
total = 0;
contador = 1;
/* fase de processamento */
while  (contador <= 10) {
        printf  ("Entre com o grau: ");
        scanf  ("%d",  &grau);
        total = total + grau;
        contador = contador + 1;
}
/* fase de terminação */
media = total / 10;
printf  ("A media da turma e %d/n", media);
return 0;  /* indica que o programa terminou corretamente */
}
```

```
Entre com o grau:     98
Entre com o grau:     7 6
Entre com o grau:     71
Entre com o grau:     87
Entre com o grau:     83
Entre com o grau:     90
Entre com o grau:     57
Entre com o grau:     79
Entre com o grau:     82
Entre com o grau:     94
A media da turma e 81
```

Fig. 3.6 Programa em C e exemplo de execução do problema de média da turma com repetição controlada por contador.



Boa prática de programação 3.6

Inicialize contadores e totais.

Observe que o cálculo da média no programa produziu um resultado inteiro. Na realidade, a soma" dos graus nesse exemplo é 817 que ao ser dividido por 10 deveria levar a 81,7, i.e., um número com ponto decimal. Veremos como lidar com tais números (chamados números de ponto flutuante) na próxima seção.

3.9 Formulando Algoritmos com Refinamento Top-Down por Etapas: Estudo de Caso 2 (Repetição Controlada por Sentinela)

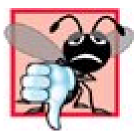
Vamos generalizar o problema do cálculo da média da turma. Considere o seguinte problema:

Desenvolva um programa para calcular a média de uma turma que processe um número arbitrário de graus cada vez que o programa for executado.

No primeiro exemplo de cálculo de média da turma, o número de graus (10) era conhecido de antemão. Nesse exemplo, não há indicação de quantos graus serão fornecidos. O programa deve processar um número arbitrário de graus. Como o programa pode saber quando deve parar de obter graus? Como ele saberá quando calcular e imprimir a média da turma?

Uma maneira de resolver esse problema é usar um valor especial chamado *valor sentinela* (também chamado *valor sinalizador*, *valor fictício, valor provisório, valor dummy ou flag)* para indicar o "final da entrada de dados". O usuário digita graus até que todos os graus válidos tenham sido fornecidos. Então o usuário digita o valor sentinela para indicar que o último grau foi fornecido. A repetição controlada por sentinela é chamada freqüentemente *repetição indefinida* porque o número de repetições não é conhecido antes de o loop começar a ser executado.

Obviamente, o valor sentinela deve ser escolhido de forma que não possa ser confundido com um valor aceitável de entrada. Como os graus no teste são normalmente inteiros não-negativos, — 1 é um valor aceitável para sentinela nesse caso. Dessa forma, a execução do programa de média da turma pode processar um fluxo de entradas como 95, 96, 75, 74, 89 e -1. A seguir, o programa calcularia e imprimiria a média da turma para os graus 95, 96, 75, 74 e 89 (- 1 é o valor sentinela, portanto ele não deve entrar no cálculo da média).



Erro comun de programação 3.6

Escolher um valor sentinela que também seja um valor aceitável de entrada de dados.

Desenvolveremos o programa de média da turma com uma técnica chamada *refinamento top-down (descendente) em etapas*, técnica essencial para o desenvolvimento de programas bem-estruturados. Começaremos com uma representação de pseudocódigo do *nível superior (topo* ou *top)* do refinamento:

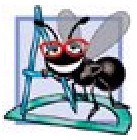
Determinar a média da turma no teste

O nível superior é uma única declaração que exprime a função global do programa. Assim, o nível superior é, na realidade, uma representação completa do programa. Infelizmente, o nível superior (como nesse caso) raramente transmite uma quantidade suficiente de detalhes para o programa em C.

Começamos agora o processo de refinamento. Dividimos o nível superior (topo) em uma série de tarefas menores e as listamos na ordem em que precisam ser realizadas. Isso resulta no *primeiro refinamento* seguinte:

Inicializar as variáveis
Obter, somar e contar as notas no teste
Calcular e imprimir a média da turma

Aqui, apenas a estrutura da seqüência foi utilizada — e os passos listados devem ser executados na ordem apresentada, um após o outro.



Observação de engenharia de software 3.3

Cada refinamento, assim como o nível superior em si, é uma especificação completa do algoritmo; é apenas o nível de detalhes que varia.

Para continuar até o próximo nível de refinamento, i.e., o *segundo refinamento,* definimos as variáveis específicas. Precisamos de um total da soma dos números, uma contagem de quantos números foram processados, uma variável para receber o valor de cada grau quando ele é fornecido e uma variável para conter a média calculada. A declaração do pseudocódigo

Inicializar as variáveis

pode ser refinada da seguinte maneira:

Inicializar o total com o valor zero Inicializar o contador com o valor zero

Observe que apenas o total e o contador precisam ser inicializados; a média das variáveis e o grau (para a média calculada e a entrada do usuário, respectivamente) não precisam ser inicializados porque seus valores serão determinados pelo processo de leitura destrutiva analisado no Capítulo 2. A instrução do pseudocódigo

Obter, somar e contar as notas no teste

exige uma estrutura de repetição (i.e., um loop) que obtenha sucessivamente cada grau. Como não sabemos de antemão quantos graus devem ser processados, usaremos uma repetição controlada por sentinela. O usuário digitará um grau válido de cada vez. Depois de o último grau válido ser digitado, o usuário digitará o valor sentinela. O programa examinará o valor fornecido após cada entrada e terminará o loop quando a sentinela for obtida. O refinamento do pseudocódigo anterior será então

Obter o primeiro grau
Enquanto o usuário ainda não fornecer a sentinela Somar este grau ao total Somar um ao contador de graus Obter o próximo grau (possivelmente a sentinela)

Observe que, no pseudocódigo, não usamos chaves em torno do conjunto de instruções que formam o corpo da estrutura *while* [que significa *enquanto,* em inglês]. Simplesmente recuamos todas as instruções sob o *while* para mostrar que elas lhe pertencem. Mais uma vez repetimos que o pseudocódigo é apenas uma ajuda informal ao desenvolvimento do programa.

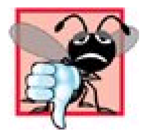
A instrução do pseudocódigo

Calcular e imprimir a média da turma

pode ser refinada da seguinte forma:

Se o contador for igual a zero
Definir a média como o total dividido pelo contador :
Imprimir a média senão
Imprimir "Nenhum grau foi fornecido"

Observe que aqui estamos tendo o cuidado de verificar a possibilidade de divisão por zero — um *erro fatal* que, se não for detectado, fará com que o programa falhe (chamado freqüentemente *"quebrar", "bombing", "crashing"* ou, informalmente, *"dar pau").* O segundo refinamento completo está mostrado na Fig. 3.7.



Erro comun de programação 3.7

Uma tentativa de dividir por zero causa um erro fatal.

Inicializar total com o valor zero Inicializar contador com o valor zero
Obter o primeiro grau
Enquanto o usuário ainda não fornecer o valor sentinela Adicionar o grau ao total variável Adicionar um ao contador de graus Obter o próximo grau (possivelmente o valor sentinela)
Se o contador for diferente de zero
Definir a média com o valor total dividido por contador
Imprimir a média senão
Imprimir "Nenhum grau foi fornecido"

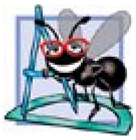
Fig. 3.7 Algoritmo do pseudocódigo que usa repetição controlada por sentinela para resolver o problema da média da turma.



Boa prática de programação 3.7

Ao realizar uma divisão por uma expressão que pode assumir o valor zero, teste explicitamente se este é o caso e trate-o adequadamente em seu programa (como imprimir uma mensagem de erro) em vez de permitir a ocorrência de um erro fatal.

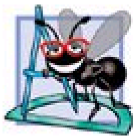
Nas Figs. 3.5 e 3.7, incluímos algumas linhas completamente em branco no pseudocódigo para melhorar sua legibilidade. Na realidade, as linhas em branco dividem esses programas em suas várias fases.



Observação de engenharia de software 3.4

Muitos programas podem ser divididos logicamente em três fases: uma fase de inicialização que inicializa as variáveis do programa; uma fase de processamento que obtém os valores dos dados e ajusta as variáveis do programa de modo correspondente; e uma fase de terminação (finalização) que calcula e imprime os resultados finais.

O algoritmo do pseudocódigo na Fig. 3.7 resolve o problema mais geral de média da turma. Esse algoritmo foi desenvolvido após apenas dois níveis de refinamento. Algumas vezes são necessários mais níveis.



Observação de engenharia de software 3.5

O programador termina o processo de refinamento top-down em etapas quando o algoritmo do pseudocódigo especificar detalhes suficientes para o programador ser capaz de converter o pseudocódigo em C. Normalmente, depois disso fica simples implementar o programa em C.

O programa em C e um exemplo de execução são mostrados na Fig. 3.8. Embora tenham sido fornecidos apenas graus inteiros, provavelmente o cálculo da média produz um resultado com ponto decimal. O tipo **int** não pode representar tal número. O programa apresenta o tipo de dados **float** para manipular números com pontos decimais (chamados *números de ponto flutuante)* e apresenta um operador especial chamado *operador de coerção (operador de conversão,* ou *cast operator)* para manipular o cálculo da média. Estes recursos são explicados em detalhes depois da apresentação do programa.

Observe a instrução composta no loop **while** na Fig. 3.8. Repetimos mais uma vez que as chaves são necessárias para que todas as quatro instruções dentro do loop sejam executadas. Sem as chaves, as

```
/* Programa de media da turma com
repetição controlada por sentinela */
#include <stdio.b>
main ( )
{
float media;                /* novo tipo de dado */
int contador,  grau, total;
/* fase de inicialização */
 total = 0; contador = 0;
/* fase de processamento */
printf("Entre com o grau,   -1 para finalizar:");
scanf("%d",  &grau);
while (grau != -1) {
total = total + grau; contador = contador + 1;
printf("Entre com o grau,  -1 para finalizar:");
scanf("%d",  &grau);
}
/* fase de terminação */
if  (contador  != 0) {
```

```
media =  (float)  total / contador;
printf ("A media da turma e %.2f", media); }
else
printf("Nenhum grau foi fornecido\n");

return 0;  /* indica que o programa terminou corretamente */
}
```

```
Entre com o grau,-1 para finalizar: 75
Entre com o grau,-1 para finalizar: 94
Entre com o grau,-1 para finalizar: 97
Entre com o grau,-1 para finalizar: 88
Entre com o grau,-1 para finalizar: 7 0
Entre com o grau,-1 para finalizar: 64
Entre com o grau,-1 para finalizar: 83
Entre com o grau,-1 para finalizar: 89
Entre com o grau,-1 para finalizar: -1
         A media da turma e 82.50
```

Fig. 3.8 Programa em C e exemplo de execução do problema de média da turma com repetição controlada por sentinela.

últimas três instruções no corpo do loop ficariam fora dele, fazendo com que o computador interpretasse o código erradamente, como se segue:

```
while  (grau != -1)
total = total + grau; contador = contador + 1;
printf("Entre com o grau,  -1 para encerrar:   ");
scanf ("%d",  &grau) ;
```

Isso ocasionaria um loop infinito se o usuário não fornecesse — 1 como primeiro grau.



Boa prática de programação 3.8

Em um loop controlado por sentinela, os pedidos de entrada de dados (prompts) devem indicar explicitamente ao usuário o valor da sentinela.

Nem sempre as médias são calculadas como valores inteiros. Freqüentemente, uma média é um valor como 7,2 ou -93,5 que contém uma parte fracionária. Esses valores são chamados números de ponto flutuante e são representados pelo tipo de dados **float.** A variável **media** é declarada do tipo **float** para conter o resultado fracionário do cálculo. Entretanto, o resultado de **total / contador** é inteiro porque **total** e **contador** são valores inteiros. Dividir dois inteiros resulta em uma *divisão inteira* na qual a parte fracionária é ignorada antes de o resultado ser atribuído a **media.** Para produzir um cálculo em ponto flutuante com valores inteiros, devemos criar valores temporários que sejam números de ponto flutuante para o cálculo. A linguagem C fornece o *operador unário de coerção (conversão)* para realizar essa tarefa. A instrução

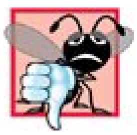
```
media =  (float)  total / contador;
```

inclui o operador de coerção **(float)** que cria uma cópia temporária em ponto flutuante de seu operando, **total.** Usar um operador dessa maneira é chamado *conversão explícita.* O valor armazenado em **total** ainda é um inteiro. Agora o cálculo consiste em um valor de ponto flutuante (a versão **float** temporária de **total)** dividido pelo valor inteiro armazenado em **contador.** O compilador C só sabe calcular expressões nas quais os tipos de dados dos operandos são idênticos. Para assegurar que os operandos sejam do mesmo tipo, o compilador realiza uma operação chamada *promoção* (também chamada *conversão implícita)* em operadores selecionados. Por exemplo, em uma expressão que contenha tipos de dados **int** e **float,** o padrão ANSI especifica que sejam feitas cópias dos operandos **int** e que elas sejam *promovidas* a **float.** Em nosso exemplo, depois de ter sido feita uma cópia de **contador** e ela ter sido promovida a **float,** o cálculo é realizado e o resultado da divisão em ponto flutuante é atribuído a **media.** O padrão ANSI fornece um conjunto de regras para promoção de operandos de diferentes tipos. O Capítulo 5 apresenta uma análise de todos os tipos padronizados de dados e sua ordem de promoção.

Os operadores de conversão estão disponíveis para qualquer tipo de dado. O operador de conversão é formado pela colocação de parênteses em torno do nome de um tipo de dado. O operador de conversão é um *operador unário,* i.e., um operador que necessita apenas de um operando. No Capítulo 2, estudamos os operadores aritméticos binários. O C também suporta versões unárias dos operadores de adição (+) e de subtração (--), de forma que o programador pode escrever expressões como **—7** ou **—5.** Os operadores de conversão são associados da direita para a esquerda e têm a mesma precedência que outros operadores unários como o unário + e o unário —. Essa precedência está um nível acima daquele dos *operadores multiplicativos* *, / e % e um nível abaixo daquele dos parênteses.

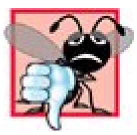
O programa da Fig. 3.8 usa o especificador de conversão %. **2 f** de **printf** para imprimir o valor de **media.** O **f** especifica que será impresso um valor de ponto flutuante. O . **2** é a *precisão* na qual o valor será exibido. Ela afirma que o valor será exibido com 2 dígitos decimais à direita da vírgula (ou ponto decimal). Se fosse utilizado o especificador de conversão %f (sem especificação da precisão), seria empregada a *precisão default* de 6 — exatamente como se fosse utilizado o especificador de conversão %.**6f**. Quando os valores em ponto flutuante são impressos com precisão, o valor impresso é *arredondado* para o número de posições decimais indicadas. O valor na memória permanece inalterado. Quando as instruções a seguir são executadas, os valores 3.45 e 3.4 são impressos.

```
printf("%.2f\n", 3.446); /*imprime 3.45 */
printf("%.1f\n",  3.446);  /*imprime 3.4 */
```



Erro comun de programação 3.8

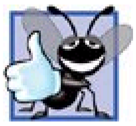
*Usar precisão em uma especificação de conversão na string de controle de formato de uma instrução **scanf** é errado. As precisões são usadas somente em especificações de conversão de **printf.***



Erro comun de programação 3.9

Usar números de ponto flutuante de uma maneira que os considere representados precisamente pode levar a resultados incorretos. Os números de ponto flutuante são representados apenas aproximadamente na maioria dos computadores.

Boa prática de programação 3.9



Não faça comparações de igualdade com valores de ponto flutuante.

Apesar do fato de que os números de ponto flutuante não são sempre "100% precisos", eles possuem várias aplicações. Por exemplo, quando falamos de uma temperatura "normal" do corpo de 36,5 não é necessário ser preciso com um grande número de dígitos. Quando vemos a temperatura em um termômetro e a lemos como 36,5, ela pode ser 36,499473210643.0 fundamental aqui é que se referir àquele número simplesmente como 36,5 está ótimo para a maioria das aplicações. Falaremos mais sobre essa questão mais adiante.

Outra maneira de obter números de ponto flutuante é através da divisão. Quando dividimos 10 por 3, o resultado é 3,33333333... com uma seqüência de 3 repetida infinitamente. O computador aloca apenas um espaço fixo para conter tal valor, portanto obviamente o valor de ponto flutuante armazenado só pode ser uma aproximação.

3.10 Formulando Algoritmos com Refinamento Top-Down por Etapas: Estudo de Caso 3 (Estruturas de Controle Aninhadas)

Vamos examinar outro problema completo. Formularemos novamente o algoritmo utilizando o pseudocódigo e o refinamento top-down por etapas, e escreveremos o programa correspondente em C. Vimos que as estruturas de controle podem ser colocadas (empilhadas) umas sobre as outras (em seqüência) da mesma forma que uma criança empilha blocos de construção. Neste estudo de caso veremos a única outra maneira existente em C para conectar estruturas de controle, isto é, o *aninhamento* de uma estrutura de controle em outra.

Considere o seguinte enunciado de um problema.

Uma faculdade oferece um curso que prepara alunos para o exame estadual de licenciamento para corretores de imóveis. No ano passado, vários alunos que completaram o curso fizeram o exame de licenciamento. Naturalmente, a faculdade deseja saber como os estudantes foram no exame. Foi pedido a você para fazer um programa que resumisse os resultados. Você recebeu uma lista de 10 desses alunos. Ao lado de cada nome está escrito um 1 se o aluno passou no exame e um 2 se ele não passou.

Seu programa deve analisar o resultado do exame da seguinte maneira:

1. *Entre com o resultado de cada teste (i.e., um 1 ou um 2). Exiba a mensagem "Entre com o resultado " na tela sempre que o programa solicitar mais um resultado de teste.*
2. *Conte o número de resultados de teste de cada tipo.*
3. *Exiba um resumo dos resultados do teste indicando o número de alunos que passaram e o número de alunos reprovados.*
4. *Se mais de 8 alunos passaram no exame, imprima a mensagem "Cobrar Taxa Escolar".*

Depois de ler cuidadosamente o enunciado do problema, fazemos as seguintes observações:

1. O programa deve processar 10 resultados do teste. Será utilizado um loop controlado por contador.
2. Cada resultado do teste é um número — seja ele 1 ou 2. Cada vez que o programa ler um resultado do teste, deve determinar se o número é 1 ou 2. Testamos se o resultado é o número 1 em nosso algoritmo. Se o número não for 1, presumimos que ele é 2. (Um exercício no final do capítulo avalia as conseqüências dessa suposição.)
3. São usados dois contadores — um para contar o número de alunos que passaram no exame e outro para contar o número de alunos reprovados.
4. Depois de o programa ter processado todos os resultados, ele deve decidir se mais de 8 alunos passaram no exame.

Vamos utilizar o refinamento top-down em etapas. Iniciamos com uma representação em pseudocódigo do nível superior do refinamento.

Analisar os resultados do exame e decidir se a taxa escolar deve ser cobrada.

Enfatizamos mais uma vez que o nível superior (topo) é uma representação completa do programa, mas é provável que se façam necessários muitos refinamentos antes que o

pseudocódigo possa ser transformado naturalmente em um programa em C. Nosso primeiro refinamento é

Inicializar as variáveis
Entrar com os dez graus no teste e contar as aprovações e reprovações
Imprimir um resumo dos resultados do exame e decidir se a taxa escolar deve ser cobrada

Aqui também, muito embora tenhamos uma representação completa de todo o programa, mais refinamentos se fazem necessários. É preciso ter contadores para registrar as aprovações e reprovações, deve ser usado um contador para controlar o processo do loop e se faz necessário ter uma variável para armazenar o valor fornecido pelo usuário. A instrução de pseudocódigo

Inicializar as variáveis

pode ser refinada da seguinte maneira

Inicializar aprovações com o valor zero Inicializar reprovações com o valor zero Inicializar o contador de alunos com o valor um

Observe que apenas os contadores e totais são inicializados. A instrução de pseudocódigo

Entrar com os dez graus no teste e contar as aprovações e reprovações

exige um loop que obtenha sucessivamente o resultado de cada exame. Aqui sabe-se de antemão que há exatamente dez resultados do exame, portanto um loop controlado por contador é adequado. Dentro do loop (i.e., *aninhado* no interior do loop) uma estrutura de seleção dupla determinará se o resultado de cada exame é uma aprovação ou uma reprovação e incrementará os contadores de maneira apropriada. O refinamento da instrução anterior do pseudocódigo fica então

Enquanto o contador de alunos for menor que ou igual a dez Obter o resultado do próximo exame
Se o aluno foi aprovado
Adicionar um a aprovações senão
Adicionar um a reprovações
Adicionar um ao contador de alunos

Preste atenção no uso de linhas em branco para destacar a estrutura *If/else (Se/senão)* e melhorar a legibilidade do programa. A instrução do pseudocódigo

Imprimir um resumo dos resultados do exame e decidir se a taxa escolar deve ser cobrada
Inicializar aprovações com o valor zero Inicializar reprovações com o valor zero Inicializar aluno com o valor um
Enquanto o contador de alunos for menor que ou igual a dez Entrar com o resultado do próximo exame
Se o aluno foi aprovado
Adicionar um a aprovações senão
Adicionar um a reprovações
Adicionar um ao contador de alunos

Imprimir o número de aprovações Imprimir o número de reprovações Se mais de oito alunos foram aprovados Imprimir "Cobrar taxa escolar"

Fig. 3.9 Pseudocódigo para o problema de resultados do exame.

pode ser refinada da seguinte maneira

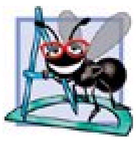
Imprimir o número de aprovações Imprimir o número de reprovações Se mais de oito alunos forem aprovados Imprimir "Cobrar taxa escolar"

O segundo refinamento completo aparece na Fig. 3.9. Observe que também são usadas linhas em branco para destacar a estrutura *while (enquanto)* e melhorar a legibilidade.
Agora esse pseudocódigo está suficientemente refinado para conversão ao C. O programa em C e exemplos de duas execuções são mostrados na Fig. 3.10. Observe que tiramos proveito de um recurso do C que permite incorporar a inicialização nas declarações. Tal inicialização ocorre em tempo de compilação.



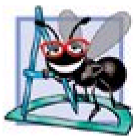
Dica de desempenho 3.1

Inicializar as variáveis durante sua declaração reduz o tempo de execução do programa.



Observação de engenharia de software 3.6

A experiência demonstrou que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Normalmente, depois de o algoritmo correto ter sido especificado, o processo de produzir um programa funcional em C fica simples.



Observação de engenharia de software 3.7

Muitos programadores escrevem programas sem usar qualquer ferramenta de desenvolvimento de programas como o pseudocódigo. Na opinião desses programadores, o objetivo final é resolver o problema em um computador, e escrever o pseudocódigo simplesmente atrasa a produção dos resultados finais.

3.11 Operadores de Atribuição

A linguagem C fornece vários operadores de atribuição para abreviar as expressões de atribuição. Por exemplo, a instrução

c = c + 3;

pode ser abreviada com o *operador de atribuição de adição* += como **c += 3 ;**

O operador += adiciona o valor da expressão à sua direita ao valor da variável à sua esquerda do operador e armazena o resultado nessa variável. Qualquer instrução da forma

variável = variável operador expressão;

```
/* Analise de resultados de exame */
#include <stdio.h>
main ( )
{
/* inicializando variáveis em declarações */
int aprovacoes = 0,  reprovacoes = 0,  aluno = 1, resultado;
/* processamento de 10 alunos; loop controlado por contador*/
while  (aluno <= 10) {
        printf("Entre com o resultado (l=aprovado,2=reprovado): ");
        scanf("%d",  fcresultado);
        if  (resultado == 1)      /* if/else no interior do while */
                aprovacoes = aprovacoes + 1;
        else
                reprovacoes = reprovacoes + 1;
        aluno = aluno + 1;
        }
printf("Aprovados %d\n", aprovacoes);
printf("Reprovações %d\n", reprovacoes);
if  (aprovacoes > 8)
        printf  ("Cobrar taxa escolar\n");
        return 0;  /* finalização correta */
}
```

```
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   2
Entre com o resultado    (l=aprovado,    2=reprovado):   2
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   2
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   1
Entre com o resultado    (l=aprovado,    2=reprovado):   2
        Aprovados 6                      Reprovados 4
```

```
Entre com o resultado      (l=aprovado, 2=reprovado):          1
Entre com o resultado      (l=aprovado,  2=reprovado):         1
Entre com o resultado      (l=aprovado,  2=reprovado):         1
Entre com o resultado      (l=aprovado, 2=reprovado):          2
Entre com o resultado      (l=aprovado, 2=reprovado):          1
Entre com o resultado      (l=aprovado, 2=reprovado):          1
Entre com o resultado      (l=aprovado, 2=reprovado);          1
Entre com o resultado      (l=aprovado,  2=reprovado):         1
Entre com o resultado      (l=aprovado,  2=reprovado):         1
Entre com o resultado      (l=aprovado, 2=reprovado):          1
          Aprovados 9                Reprovados 1
Cobrar taxa escolar
```

Fig. 3.10 Programa em C e exemplos de execução do problema de resultados do exame.

onde operador é um dos operadores binários +, —, *, / ou % (ou outros que analisaremos no Capítulo 10) pode ser escrita na forma

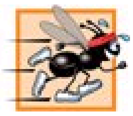
variável operador = expressão;

Assim, a atribuição **c +=** 3 adiciona 3 a **c.** A Fig. 3.11 mostra os operadores aritméticos de atribuição, expressões usando esses operadores e explicações.



Dica de desempenho 3.2

*Uma expressão com um operador de atribuição (como em **c += 3)** é compilada mais rapidamente que a expressão equivalente expandida **(c = c + 3)** porque na primeira expressão **c** é avaliado apenas uma vez, enquanto na segunda expressão ele é avaliado duas vezes.*



Dica de desempenho 3.4

Muitas das dicas de performance que mencionamos neste texto resultam em melhorias insignificantes, portanto o leitor pode se ver tentado a ignorá-las. Deve ser ressaltado aqui que é o efeito acumulativo de todas essas otimizações de performance que pode tornar o programa significativamente mais rápido. Além disso, percebe-se uma melhora muito grande quando um aperfeiçoamento supostamente insignificante é colocado em um loop que pode se repetir muitas vezes.

3.12 Operadores de Incremento e Decremento

Operador de atribuição	Exemplo de expressão	Explicação	Atribui
Admita: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	**c += 7**	**c = c + 7**	**10** a **c**
-=	**d -= 4**	**d = d - 4**	**1** a **d**
***=**	**e *= 5**	**e = e * 5**	**20** a **e**
/=	**f /= 3**	**f = f / 3**	**2** a **f**
%=	**g %= 9**	**g = g % 9**	**3** a **g**

Fig. 3.11 Operadores aritméticos de atribuição.

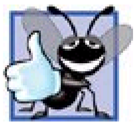
Operador	Exemplo de expressão	Explicação
++	**++a**	Incrementa **a** de 1 e depois usa o novo valor de **a** na expressão onde **a** se localiza.
++	**a**++	Usa o valor atual de **a** na expressão onde **a** se localiza e depois incrementa **a** de 1.
--	**--b**	Decrementa **b** de 1 e depois usa o novo valor de **b** na expressão onde **b** se localiza.
--	**b--**	Usa o valor atual de **b** na expressão onde **b** se localiza e depois decrementa **b** de 1.

Fig. 3.12 Operadores de incremento e decremento,

A linguagem C também fornece o *operador unário de incremento, ++,* e o *operador unário de decremento,* , que estão resumidos na Fig. 3.12. Se uma variável **c** é incrementada de 1, o operador de incremento ++ pode ser usado no lugar das expressões **c = c + 1** ou **c =+ 1.** Se os operadores de incremento ou decremento forem colocados antes de uma variável, eles são chamados *operadores de pré-incremento* ou *pré-decremento.* Se os operadores de incremento ou decremento forem colocados depois de uma variável, são chamados *operadores de pós-incremento* ou *pós-decremento,* respectivamente. Pré-incrementar (pré-decrementar) uma variável faz com que ela seja incrementada (ou decrementada) de 1 e depois seu novo valor seja usado na expressão onde ela aparece. Pós-incrementar (pós-decrementar) a variável faz com que seu valor atual seja utilizado na expressão onde ela aparece e depois seu valor seja incrementado (decrementado) de 1.

O programa na Fig. 3.13 mostra a diferença entre as versões do operador ++ para pré-incrementar e pós-incrementar. Pós-incrementar a variável **c** faz com que ela seja incrementada depois de ser utilizada na instrução **printf.** Pré-incrementar a variável **c** faz com que ela seja incrementada antes de ser utilizada na instrução **printf.**

O programa exibe o valor de **c** antes e depois de o operador ++ ser usado. O operador de decremento funciona de modo similar.



Boa prática de programação 3.10

Os operadores unários devem ser colocados imediatamente após seus operandos sem nenhum espaço intermediário.

As três instruções de atribuição da Fig. 3.10

```
aprovacoes = aprovacoes + 1;
reprovacoes = reprovacoes + 1;
aluno = aluno + 1;
```

podem ser escritas mais resumidamente com os operadores de atribuição da seguinte forma

```
aprovacoes += 1; reprovacoes += 1;
aluno += 1;
```

ou com os operadores de pré-incremento da seguinte forma

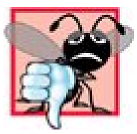
```
++aprovacoes;
++reprovacoes;
++aluno;
```

ou com os operadores de pós-incremento da seguinte forma

```
aprovacoes++; reprovacoes++;
aluno++;
```

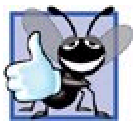
É importante destacar aqui que quando uma variável é incrementada ou decrementada em uma instrução isolada, as formas de incrementar e decrementar causam o mesmo efeito. Somente quando uma variável aparece no contexto de uma expressão maior é que os efeitos de pré-incrementar e pós-incre-mentar serão diferentes (o mesmo acontece com pré-decrementar e pós-decrementar).

Somente um nome simples de variável pode ser usado como operando de um operador de incremento ou decremento.



Erro comum de programação 3.10

Tentar usar o operador de incremento ou decremento em uma expressão que não seja um nome simples de variável, e.g., escrever ***++ (x+1)*** *é um erro de sintaxe.*



Boa prática de programação 3.11

Geralmente, o padrão ANSI não especifica a ordem em que os operandos de um operador serão processados (embora vejamos algumas exceções para isso para alguns operadores no Capítulo 4). Portanto, o programador deve evitar o uso de instruções com operadores de incremento ou decremento nas quais uma determinada variável que esteja sendo incrementada apareça mais de uma vez.

A tabela da Fig. 3.14 mostra a precedência e a associatividade dos operadores apresentados até aqui. Os operadores são mostrados de cima para baixo na ordem decrescente de precedência. A segunda colima descreve a associatividade dos operadores em cada nível de precedência. Observe que o operador condicional (? :), os operadores unários de incremento (++), decremento (--), adição (+), subtração (-) e conversão, e os operadores de atribuição =, +=, -=,*=,/= e %= são associados da direita para a esquerda. A terceira coluna fornece o nome dos vários grupos de operadores. Todos os outros operadores na Fig. 3.14 são associados da esquerda para a direita.

```
/* Pre-incrementando e pos-incrementando */
#include <stdio.h>
main ()
{
int c ; c = 5;
printf("%d\n",  c);
printf("%d\n", c++); /* pos-incremento */
printf ("%d\n\n",  c) ;
c = 5;
printf("%d\n", c);
printf("%d\n", ++c); /* pre-incremento */
printf("%d\n",  c);
return 0;      /* finalização correta */
}
```

```
5 5 6
5 6 6
```

Fig. 3.13 Mostrando a diferença entre pré-incrementar e pós-incrementar.

Operadores	Associatividade	Tipo
( )	Esquerda para a direita	Parênteses
++ -- + - (tipo)	Direita para esquerda	Unário
* ? %	Esquerda para a direita	Multiplicativo
+ -	Esquerda para a direita	Aditivo
< <= > >=	Esquerda para direita	Relacional
== !=	Esquerda para direita	Igualdade
?:	Direita para esquerda	Condicional
= += -= *= /= %=	Direita para esquerda	atribuição

Fig. 3.14 Precedência entre os operadores encontrados até agora no texto.

Resumo

- A solução de qualquer programa computacional envolve a realização de uma série de ações em uma ordem especificada. Um procedimento para resolver um problema em função das ações a serem executadas e a ordem na qual elas devem ser realizadas é chamado algoritmo.
- Especificar a ordem em que as instruções devem ser executadas em um programa computacional é chamado controle do programa.
- O pseudocódigo é uma linguagem artificial e informal que ajuda os programadores no desenvolvimento de algoritmos. Ele é similar à língua portuguesa cotidiana. Os programas em pseudocódigo não são na verdade executados em computadores. Em vez disso, o pseudocódigo simplesmente ajuda o programador a "pensar detalhadamente" no programa antes de tentar escrevê-lo em uma linguagem de programação como o C.
- O pseudocódigo consiste exclusivamente em caracteres, portanto os programadores podem digitar programas em pseudocódigo no computador, editá-los e salvá-los.
- O pseudocódigo consiste apenas em instruções executáveis. Declarações são mensagens para o compilador que lhe dizem os atributos das variáveis e lhe pedem que seja reservado espaço para elas.
- Usa-se uma estrutura de seleção para fazer uma escolha entre linhas de ação alternativas.
- A estrutura de seleção **if** executa uma ação indicada apenas quando a condição é verdadeira.
- A estrutura de seleção **if/else** especifica ações diferentes a serem executadas conforme a condição seja verdadeira ou falsa.
- Uma estrutura de seleção **if/else** aninhada pode testar muitas situações diferentes. Se mais de uma condição for verdadeira, apenas as instruções após a primeira condição verdadeira serão executadas.
- Sempre que mais de uma instrução devam ser executadas onde devia haver apenas uma instrução simples, essas instruções devem estar entre chaves, formando uma instrução composta. Uma instrução composta pode ser colocada em qualquer lugar onde pode estar uma instrução simples.
- Indica-se uma instrução vazia, utilizada para informar que nenhuma ação deve ser realizada, colocando ponto-e-vírgula (;) onde a instrução normalmente deveria estar.
- Uma estrutura de repetição especifica que uma ação deve ser repetida enquanto uma determinada condição for verdadeira.
- O formato da estrutura de repetição **while** é

  while *(condição) instrução*

A instrução (ou instrução composta ou bloco) contida na estrutura de repetição **while** constitui o corpo do loop.

- Normalmente, uma ação especificada no corpo de um **while 1 e** deve fazer com que, posteriormente, a condição se torne falsa. Caso contrário, o loop nunca terminará — um erro conhecido como loop infinito.
- O loop controlado por contador usa uma variável como contador para determinar quando um loop deve terminar.
- Um total é uma variável que acumula a soma de uma série de números. Normalmente, os totais devem ser inicializados com o valor zero antes de um programa ser executado.
- Um fluxograma é uma representação gráfica de um algoritmo. Os fluxogramas são desenhados usando determinados símbolos especiais, como retângulos, elipses, losangos e pequenos círculos, conectados por setas chamadas linhas de fluxo. Os símbolos indicam as ações a serem realizadas. As linhas de fluxo indicam a ordem em que as ações devem ser executadas.

• O símbolo em forma de elipse, também chamado símbolo de terminação, indica o início e o fim de um algoritmo.
• O símbolo em forma de retângulo, também chamado símbolo de ação, indica qualquer tipo de cálculo ou operação de entrada/saída. Os símbolos retangulares correspondem a ações realizadas normalmente por operações de atribuição ou a operações de entrada/saída realizadas normalmente pelas funções da biblioteca padrão como **printf** e **scanf.**
• O símbolo em forma de losango, também chamado símbolo de decisão, indica que deve ser tomada uma decisão. O símbolo de decisão contém uma expressão que pode ser verdadeira ou falsa. Do símbolo saem duas linhas de fluxo. Uma delas indica a direção a ser tomada quando a condição é verdadeira; a outra indica a direção a ser tomada quando a condição é falsa.
• Um valor que contém uma parte fracionária é chamado número de ponto flutuante e é representado pelo tipo de dado **float.**
• Dividir dois inteiros resulta em uma divisão inteira na qual a parte fracionária do cálculo é ignorada (i.e., truncada).
• A linguagem C fornece o operador unário de conversão **(float)** para criar uma cópia em ponto flutuante de seu operando. Usar um operador de conversão (coerção) desta forma é chamado conversão explícita. Os operadores de conversão estão disponíveis para qualquer tipo de dado.
• O compilador C só sabe como calcular expressões nas quais os tipos de dados dos operandos sejam idênticos. Para assegurar que os operandos sejam do mesmo tipo, o compilador realiza uma operação chamada promoção (também chamada conversão implícita) em operandos selecionados. O padrão ANSI especifica que sejam feitas cópias dos operandos **int** e promovidas a **float.** O padrão ANSI fornece um conjunto de regras para a promoção de operandos de diferentes tipos.
• Os valores de ponto flutuante são exibidos com um número específico de dígitos após o ponto decimal, através do especificador de conversão ***%t*** em uma instrução **printf.** O valor **3 .456** é exibido como **3.46** com o especificador de conversão ***%*.2f**. Se o especificador de conversão %f for usado (sem especificação da precisão), a precisão 6 é adotada.
• A linguagem C fornece vários operadores de atribuição que ajudam a abreviar determinados tipos comuns de expressões aritméticas de atribuição. Esses operadores são: +=, -=, *=, /= e %=. Em geral, qualquer instrução da forma
variável = variável operador expressão;
onde **operador** é um dos operadores +, -, *, / e %, pode ser escrita na forma
variável operador = expressão;
• A linguagem C fornece o operador de incremento, ++, e o operador de decremento, - -, para incrementar ou decrementar uma variável de 1. Esses operadores podem ser colocados antes ou após uma variável. Se o operador for colocado antes de uma variável, esta é inicialmente incrementada ou decrementada de 1 e depois é utilizada na expressão. Se o operador for colocado depois de uma variável, esta é usada na expressão e depois é incrementada ou decrementada de 1.

Terminologia

ação
algoritmo
arredondamento
bloco
"bombing"
caracteres de espaço
condição de término
contador
controle do programa
conversão explícita
conversão implícita
corpo de um loop
"crashing"
dar pau
decisão
divisão inteira
divisão por zero
eliminação de **goto**
erro dc sintaxe
erro fatal
erro lógico
erro não fatal
estrutura dc controle
estrutura de repetição **while**
estrutura de seleção dupla
estrutura de seleção **if**
estrutura de seleção **if/else**
estrutura de seleção múltipla
estrutura dc seleção simples
estrutura de seqüência
estruturas de controle
aninhadas
estruturas de controle
empilhadas
estruturas dc repetição
estruturas de seleção
estruturas de única
entrada/única saída
estruturas **if/else** aninhadas
etapas
execução seqüencial
fase de inicialização
fase de processamento
fase de terminação
"final da entrada de dados"
flag
float
fluxograma
inicialização
instrução composta
instrução **goto**
instrução vazia (;)
linha de fluxo
loop infinito
looping nível superior número de ponto
flutuante operador condicional (? :)
operador de coerção operador de
conversão operador de decremento (--)
operador de incremento (++) operador de
pós-decremento operador de pós-
incremento operador de pré-decremento
operador de pré-incremento operador
ternário
operadores aritméticos de atribuição: +=,
-=, *+=, /= *e%=*
operadores multiplicativos
ordem das ações
passos
precisão
precisão default
primeiro refinamento
programação estruturada
promoção
pseudocódigo
quebrar
refinamento em etapas refinamento top-
down em etapas repetição
repetição controlada por contador
repetição definida
repetição indefinida
segundo refinamento
seleção
símbolo de ação
símbolo de decisão
símbolo de fim
símbolo de seta
símbolo de terminação
símbolo em forma de elipse
símbolo em forma de losango
símbolo em forma de retângulo
top
total
transferência de controle
truncamento
valor "lixo"
valor dummy
valor fictício
valor provisório
valor sentinela
valor sinalizador

Erros Comuns de Programação

3.1 Esquecer de uma ou ambas as chaves que limitam uma instrução composta.

3.2 Colocar um ponto e vírgula (;) depois da condição em uma estrutura **if** leva a um erro lógico em estruturas **if** de seleção simples e um erro de sintaxe em estruturas **if** de seleção dupla.

3.3 Fornecer no corpo de uma estrutura **while** uma ação que posteriormente não torna falsa a condição no **while.** Normalmente, tal estrutura de repetição nunca terminará — este erro é chamado "loop infinito".

3.4 Digitar a palavra-chave **while** com letra **W** (maiúscula) como em **While** (lembre-se de que o C é uma linguagem que faz distinção entre letras maiúsculas e minúsculas). Todas as palavras reservadas do C como **while, if** e **else** contêm apenas letras minúsculas.

3.5 Se um contador ou total não for inicializado, provavelmente os resultados de seu programa estarão incorretos. Esse é um exemplo de erro lógico.

3.6 Escolher um valor sentinela que também seja um valor aceitável de entrada de dados.

3.7 Uma tentativa de dividir por zero causa um erro fatal.

3.8 Usar precisão em uma especificação de conversão na string de controle de formato de uma instrução **scanf** é errado. As precisões são usadas somente em especificações de conversão de **printf.**

3.9 Usar números de ponto flutuante de uma maneira que os considere representados precisamente pode levar a resultados incorretos. Os números de ponto flutuante são representados apenas aproximadamente na maioria dos computadores.

3.10 Tentar usar o operador de incremento ou decremento em uma expressão que não seja um nome simples de variável, e.g., escrever ++ **(x**+1) é um erro de sintaxe.

Práticas Recomendáveis de Programação

3.1 Aplicar consistentemente as convenções para os recuos aumenta em muito a legibilidade do programa. Sugerimos valores fixos de aproximadamente 1/4 da polegada ou três espaços em branco em cada nível de recuo.

3.2 Freqüentemente, o pseudocódigo é usado para "elaborar" um programa durante o processo de projeto do programa. Depois o programa em pseudocódigo é convertido para o C.

3.3 Aplique recuos nas instruções em ambas as partes da estrutura **if/else.**

3.4 Se houver vários níveis de recuos, o recuo de cada nível deve ter a mesma quantidade adicional de espaços.

3.5 Alguns programadores preferem digitar as chaves inicial e final de instruções compostas antes de digitar cada uma das instruções incluídas nessas chaves. Isso ajuda a evitar a omissão de uma ou ambas as chaves.

3.6 Inicialize contadores e totais.

3.7 Ao realizar uma divisão por uma expressão que pode assumir o valor zero, teste explicitamente se este é o caso e trate-o adequadamente em seu programa (como imprimir uma mensagem de erro) em vez de permitir a ocorrência de um erro fatal.

3.8 Em um loop controlado por sentinela, os pedidos de entrada de dados (prompts) devem indicar explicitamente ao usuário o valor da sentinela.

3.9 Não faça comparações de igualdade com valores de ponto flutuante.

3.10 Os operadores unários devem ser colocados imediatamente após seus operandos sem nenhum espaço intermediário.

3.11 Geralmente, o padrão ANSI não especifica a ordem em que os operandos de um operador serão processados (embora vejamos algumas exceções para isto para alguns operadores no Capítulo 4). Portanto, o programador deve evitar o uso de instruções com operadores de incremento ou decremento nas quais uma determinada variável que esteja sendo incrementada apareça mais de uma vez.

Dicas de Performance

3.1 Inicializar as variáveis durante sua declaração reduz o tempo de execução do programa.

3.2 Uma expressão com um operador de atribuição (como em **c +=** 3) é compilada mais rapidamente que a expressão equivalente expandida **(c = c +** 3) porque na primeira expressão **c** é avaliado apenas uma vez, enquanto na segunda expressão ele é avaliado duas vezes.

3.3 Muitas das dicas de performance que mencionamos neste texto resultam em melhorias insignificantes, portanto o leitor pode se ver tentado a ignorá-las. Deve-se ressaltar aqui que é o efeito acumulativo de todas essas otimizações de performance que pode tornar o programa significativamente mais rápido. Além disso, percebe-se uma melhora muito grande quando um aperfeiçoamento supostamente insignificante é colocado em um loop que pode se repetir muitas vezes.

Observações de Engenharia de Software

3.1 Uma instrução composta pode ser colocada em qualquer lugar de um programa onde pode ser colocada uma instrução simples.

3.2 Da mesma forma que uma instrução composta pode ser colocada em qualquer local onde uma instrução simples pode estar, também é possível não ter instrução alguma, i.e., uma instrução vazia. A instrução vazia é representada colocando um ponto-e-vírgula (;) onde normalmente a instrução deveria estar.

3.3 Cada refinamento, assim como o nível superior em si, é uma especificação completa do algoritmo; é apenas o nível de detalhes que varia.

3.4 Muitos programas podem ser divididos logicamente em três fases: uma fase de inicialização que inicializa as variáveis do programa; uma fase de processamento que obtém os valores dos dados e ajusta as variáveis do programa de modo correspondente; e uma fase de terminação (finalização) que calcula e imprime os resultados finais.

3.5 O programador termina o processo de refinamento top-down em etapas quando o algoritmo do pseudocódigo especificar detalhes suficientes para o programador ser capaz de converter o pseudocódigo em C. Normalmente, depois disso fica simples implementar o programa em C.

3.6 A experiência demonstrou que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Normalmente, depois de o algoritmo correto ter sido especificado, o processo de produzir um programa funcional em C fica simples.

3.7 Muitos programadores escrevem programas sem usar qualquer ferramenta de desenvolvimento de programas como o pseudocódigo. Na opinião desses programadores, o objetivo final é resolver o problema em um computador, e escrever o pseudocódigo simplesmente atrasa a produção dos resultados finais.

Exercícios de Revisão

3.1 Responda às seguintes perguntas.
a) Um procedimento para resolver um problema em termos das ações a serem executadas e a ordem em que elas devem ser realizadas é chamado um ________________.
b) Especificar a ordem de execução das instruções pelo computador é chamado __________.
c) Todos os programas podem ser escritos em termos de três estruturas de controle ________________, ________________ e ________________.
d) A estrutura de seleção ________________ é usada para executar uma ação quando uma condição for verdadeira e outra ação quando a condição for falsa.
e) Várias instruções agrupadas entre chaves ({ e }) são chamadas ________________.
f) A estrutura de repetição ________________ especifica que uma instrução ou grupo de instruções deve ser executado repetidamente enquanto uma condição permanecer verdadeira.
g) A repetição de um conjunto de instruções um número específico de vezes é chamada repetição ________________.
h) Quando não se sabe de antemão quantas vezes um conjunto de instruções será repetido, um valor ________________ pode ser usado para finalizar a repetição.

3.2 Escreva quatro instruções diferentes em C que adicionem 1 a uma variável inteira **x.** ***33***
Escreva uma instrução simples em C que faça cada uma das tarefas seguintes:
a) Atribua a soma de **x** e **y** a **z** e incremente o valor de **x** de 1 depois do cálculo.
b) Multiplique a variável **produto** por 2 usando o operador *=.
c) Multiplique a variável **produto** por 2 usando os operadores = e *.
d) Verifique se o valor da variável **contagem** é maior que 10. Se for, imprima **"Contagem maior que 10".**
e) Decremente a variável **x** de 1 e depois a subtraia da variável **total.**
f) Adicione a variável **x** à variável **total** e depois decremente a variável **x** de 1.
g) Calcule o resto depois de **q** ser dividido por **divisor** e atribua o resultado a **q.** Escreva essa instrução de duas maneiras diferentes.
h) Imprima o valor **123 .4567** com **2** dígitos de precisão. Que valor é impresso?
i) Imprima o valor de ponto flutuante **3.14159** com três dígitos à direita do ponto decimal. Que valor é impresso?

3.3 Escreva uma instrução em C que realize as seguintes tarefas.
a) Declare as variáveis **soma** e **x** do tipo **int.**
b) Inicialize a variável **x** com o valor **1.**
c) Inicialize a variável **soma** com o valor **0.**
d) Adicione a variável **x** à variável **soma** e atribua o resultado à variável **soma.**
e) Imprima **"A soma e** :" seguido do valor da variável **soma.**

3.4 Combine as instruções escritas no Exercício 3.4 em um programa que calcule a soma dos inteiros de 1 a 10. Use a estrutura **while** para fazer um loop através das instruções de cálculo e incremento. O loop deve ter minar quando o valor de **x** se tornar 11.

3.5 Determine os valores de cada variável após o cálculo ser realizado. Admita que no início da execução de cada instrução todas as variáveis possuem o valor 5.
a) **produto *= x++;**
b) **resultado = ++x + x;**

3.6 Escreva instruções simples em C que
a) Obtenha uma variável inteira **x** com **scanf.**
b) Obtenha uma variável inteira y com **scanf.**
c) Inicialize a variável inteira **i** com o valor **1.**

d) Inicialize a variável inteira **potência** com o valor **1.**
e) Multiplique a variável **potência** por **x** e atribua o resultado a **potência.**
f) Incremente a variável y de **1.**
g) Verifique se y é menor que ou igual a **x.**
h) Forneça a variável inteira **potência** com **printf.**

3.7 Escreva um programa em C que use as instruções do Exercício 3.7 para calcular **x** elevado à potência y. O programa deve ter a estrutura dc controle de repetição **while.**

3.8 Identifique c corrija os erros em cada um dos itens seguintes:
a) **while (c <= 5) <**
produto *= c;
b) **scanf("%.4f", &valor);**
c) **if (gênero == 1)**
printf("Feminino\n") else;
printf("Masculino\n");

3.9 O que há de errado com a seguinte estrutura de repetição **while?**

```
while  (z >= 0)
        soma += z;
```

Respostas dos Exercícios de Revisão

3.1 a) Algoritmo, b) Controle do programa, c) Seqüência, seleção, repetição, d) **if/else.** e) Instrução composta, f) **while.** g) Controlada por contador, h) Sentinela.

3.2 **x = x + 1; x + = 1 ;**
++x; x+-;

3.3 a) **z = x++** + y;
b) **produto *= 2;**
c) **produto = produto * 2;**
d) **if (contagem > 10)**
printf("Contagem maior que 10.\n");
e) **total -= --x;** 0 **total += x--;** g) **q %= divisor;**
q = q % divisor; . h) **printf("%.2f", 123.4567); 123 .46** é exibido, i) **printf("%.3f", 3.14159); 3 .142** é exibido.

3.4 a) **int soma, x;**
b) **x = 1;**
c) **soma = 0;**
d) **soma += x;** ou **soma = soma + x;**
e) **printf("A soma e : %d\n", soma);**

3.5
```
/* Calcula a soma dos inteiros de 1 a 10 */
#include <stdio.h>
main() {
int soma, x;
x = 1;
soma = 0;
while  (x <= 10) {
        soma <= x; + +x;
}
printf("A soma e  : %ã\n", soma);
}
```

3.6 a) produto = 25, x = 6;
b) resultado = 12, x = 6;

3.7 a) **scanf("%d", &x) ;**
b) **scanf("%d", &y) ;**
c) **i =** 1;
d) **potência = 1;**
e) **potência *= x;**
f) **y++;**
g) **if (y <= x)**
h) **printf(*"%d"*, potência);**

3.8
```
/* eleva x a potência y */
#include <stdio.h>
main() {
int x,y,i,potência;
i = 1;
potência = 1;
```

```
scanf("%d",  &x) ;
scanf("%d", &y);
while (i <= y) {
        potência *= x; + + i;
}
printf{"%d", potência);
return 0;
}
```

3.9 a) Erro: Faltando a chave direita para fechamento do corpo do **while.**
Correção: Adicionar a chave direita após a instrução + **+ c;**
b) Erro: Precisão usada em uma especificação de conversão **scanf.** Correção: Remover . **4** da especificação de conversão.
c) Erro: Ponto-e-vírgula após o segmento **else** da estrutura **if/else** resulta em um erro lógico. O segundo **printf** será sempre executado.
Correção: Remova o ponto-e-vírgula após o **else.**

3.10 O valor da variável **z** nunca é modificado na estrutura **while.** Portanto, é criado um loop infinito. Para evitar loops infinitos, **z** deve ser decrementado para que posteriormente se torne 0.

Exercícios

3.11 Identifique e corrija os erros em cada um dos itens seguintes (Nota: pode haver mais de um erro em cada conjunto de linhas de código):

a) **if (idade >= 65);**
printf("Idade e maior que ou igual a 65\n"); else
printf("Idade e menor que 65\n");

b) **int x =** 1, **total;**
while (x <= 10) { **total += x; + +x;**
}

c) **While (x** <= 100)
total += x;
+ +x;

d) **while (y** > 0) {
printf("%d\n", y); ++y;
}

3.12 Preencha as lacunas em cada um dos itens seguintes:
a) A solução de qualquer problema envolve a realização de uma série de ações em uma específica, §
b) Um sinônimo para procedimento (procedure) é .
c) Uma variável que acumula a soma de vários números é um .
d) O processo de definir valores específicos a determinadas variáveis no início de um programa é chamado |
e) Um valor especial usado para indicar o "final da entrada de dados" é chamado um valor ______________________, _______________,_________ou_______.
f) Um é uma representação gráfica de um algoritmo.
g) Em um fluxograma, a ordem em que os passos devem ser realizados é indicada pela
h) O símbolo de terminação indica o e o de qualquer algoritmo.
i) Os símbolos retangulares correspondem aos cálculos realizados normalmente por instruções de 1
e operações de entrada/saída realizadas normalmente por chamadas às funções da biblioteca padrão.
j) O item escrito no interior de um símbolo de decisão é chamado .

3.13 O que o seguinte programa imprime? **ttinclude <stdio.h>**

```
main() {
int x = 1,  total = 0, y;
while  (x <= 10) { y = x * x printf("%ã\n", y); total += y); ++x;
}
printf("O total e %d\n", total); return 0;
}
```

3.14 Escreva uma instrução simples em pseudocódigo que indique cada uma das ações seguintes:

a) Exiba a mensagem **"Entre com dois inteiros".**
b) Atribua a soma das variáveis **x, y** e **z** à variável **p.**
c) Teste a seguinte condição em uma estrutura de seleção **if/else:** O valor atual da variável **m** é maior que duas vezes o valor atual da variável **v.**
d) Obtenha os valores das variáveis **s, r** e **t** a partir do teclado.

3.15 Formule um algoritmo em pseudocódigo que faça o seguinte:
a) Obtenha dois números a partir do teclado, calcule a soma dos números e exiba o resultado.
b) Obtenha dois números a partir do teclado, determine o maior (se houver) e o exiba.
c) Obtenha uma série de números positivos a partir do teclado, determine sua soma e a exiba. Admita que o usuário deve digitar o valor sentinela — 1 para indicar.o "final da entrada de dados".

3.16 Determine se cada uma das sentenças seguintes é verdadeira ou falsa. Se a sentença for falsa, explique por quê.

a) A experiência demonstrou que a parte mais difícil de resolver um problema em computador é produzir um programa em C.
b) O valor sentinela deve ser tal que não possa ser confundido com um valor válido de dados.
c) As linhas de fluxo indicam as ações a serem realizadas.
d) As condições escritas no interior dos símbolos de decisão sempre contêm operadores aritméticos (i.e., +, -, *, / e %).
e) No refinamento top-down em etapas, cada refinamento é uma representação completa do algoritmo.

Para os Exercícios 3.17 a 3.21, realize cada um dos seguintes passos:
1. Leia a definição do problema.
2. Formule o algoritmo usando o pseudocódigo e o refinamento top-down em etapas.
3. Escreva um programa em C.
4. Teste, depure e execute o programa em C.

3.17 Tendo em vista o alto preço da gasolina, os motoristas estão preocupados com a quilometragem percorrida por seus automóveis. Um motorista fez o controle recompletando várias vezes o tanque e registrando os quilômetros percorridos e os litros de gasolina necessários para encher o tanque. Desenvolva um programa em C que receba como dados a quilometragem dirigida e os litros usados para recompletar o tanque. O programa deve calcular e exibir a quilometragem por litro obtida para cada recompletamento. Depois de processar todas as informações, o programa deve calcular e exibir a média de quilômetros por litro obtida para todos os recompletamentos.

```
Entre com os litros consumidos  (-1 para finalizar): 12.8
Entre com os km percorridos: 287
A taxa km/litro para esse tanque foi 22.421875

Entre com os litros consumidos  (-1 para finalizar): 10.3
Entre com os km percorridos: 200
A taxa km/litro para esse tanque foi 19.417475

Entre com os litros consumidos (-1 para finalizar): 5
Entre com os km percorridos: 120
A taxa km/litro para esse tanque foi 24.000000

Entre com os litros consumidos  (-1 para finalizar): -1
A taxa total de km/litro foi 21.601423
```

3.18 Desenvolva um programa em C que determine se um cliente de uma loja de departamentos excedeu o limite de crédito de sua conta. Os seguintes dados de cada cliente estão disponíveis:

1. Número da conta
2. Saldo devedor no início do mês
3. Total de itens cobrados do cliente no mês em questão
4. Total de créditos aplicados à conta do cliente no mês cm questão.
5. Limite de crédito permitido.

O programa deve receber esses dados, calcular o novo saldo (= saldo devedor inicial + cobranças — créditos) e determinar se o novo saldo supera o limite de crédito do cliente. Para os clientes cujo limite de crédito foi excedido, o programa deve exibir o número da conta do cliente, o limite de crédito e a mensagem **"Limite de Crédito Excedido".**

```
Entre com o numero da conta (-1 para finalizar): 100
Entre com o saldo inicial: 5394.78
Entre com o total de cobranças: 1000.00
Entre com o total de créditos: 500.00
Entre com o limite de credito: 5500.00
Conta: 100
Limite de credito: 5500.00
Saldo: 5894.78
Limite de Credito Excedido.

Entre com o numero da conta (-1 para finalizar): 200
Entre com o saldo inicial: 1000.00
Entre com o total de cobranças: 123.45
Entre com o total de créditos: 321.00
Entre com o limite de credito: 1500.00

Entre com o numero da conta (-1 para finalizar): 300
Entre com o saldo inicial: 500.00
Entre com o total de cobranças: 274.73
Entre com o total de créditos: 100.00
Entre com o limite de credito: 800.00
Entre com o numero da conta (-1 para finalizar): -1
```

3.19 Uma grande companhia química paga seus vendedores por comissão. Os vendedores recebem $200 por semana mais 9 por cento de suas vendas brutas naquela semana. Por exemplo, um vendedor que vender o equivalente a $5000 em produtos químicos em uma semana recebe $200 mais 9 por cento de $5000, ou um total de $650. Desenvolva um programa em C que receba as vendas brutas de cada vendedor na última semana, calcule seu salário e o exiba. Processe os dados de um vendedor de cada vez.

```
Entre com a venda em dólares (-1 para finalizar): 5000.00
Salário: $650.00

Entre com a venda em dólares (-1 para finalizar): 1234.56
Salário: $311.11

Entre com a venda em dólares (-1 para finalizar): 1088.89
Salário: $298.00

Entre com a venda em dólares  (-1 para finalizar): -1
```

3.20 Os juros simples em um empréstimo são calculados pela fórmula
juros = capital * taxa * dias / 365
A fórmula anterior admite que **taxa** c a taxa anual de juros e portanto inclui a divisão por 365 (dias). Desenvolva um programa em C que receba os valores de **capital, taxa** e **dias** de vários empréstimos, calcule os juros simples de cada empréstimo e os exiba, usando a fórmula anterior.

```
Entre com o valor do empréstimo (-1 para finalizar): 1000.00
Entre com a taxa de juros:   .1
Entre com o periodo do empréstimo em dias: 365
O valor dos juros é: $100.00

Entre com o valor do empréstimo (-1 para finalizar): 1000.00
Entre com a taxa de juros: .08375
Entre com o periodo do empréstimo em dias: 224
O valor dos juros é: $51.40

Entre com o valor do empréstimo (-1 para finalizar): 10000.00
Entre com a taxa de juros: .09
Entre com o periodo do empréstimo em dias: 1460
O valor dos juros e: $3600.00

Entre com o valor do empréstimo (-1 para finalizar): -1
```

3.21 Desenvolva um programa em C que determine o pagamento bruto de cada um de vários empregados. A companhia paga o valor de uma "hora normal" pelas primeiras 40 horas trabalhadas de cada empregado e paga o valor de uma "hora extra" (uma vez e meia a hora normal) para cada hora trabalhada depois de completadas as primeiras 40 horas. Você recebeu uma lista de empregados da companhia, o número de horas que cada empregado trabalhou durante a semana passada e o valor da "hora normal" de cada empregado. Seu programa deve receber essas informações de cada empregado além de determinar e exibir o pagamento bruto do empregado.

```
Entre com o numero de horas trabalhadas (-1 para finalizar): 39
Entre com o valor da hora normal do trabalhador C$00.00): 10.00
Salário: $390.00

Entre com o numero de horas trabalhadas (-1 para finalizar): 40
Entre com o valor da hora normal do trabalhador ($00.00): 10.00
Salário: $400.00

Entre com o numero de horas trabalhadas (-1 para finalizar): 41
Entre com o valor da hora normal do trabalhador ($00.00): 10.00
Salário: 415.00

Entre com o numero de horas trabalhadas (-1 para finalizar): -1
```

3.22 Escreva um programa em C que demonstre a diferença entre pré-decrementar e pós-decrementar usando o operador de decremento - -.

3.23 Escreva um programa em C que utilize um loop para imprimir os números de 1 a 10, lado a lado na mesma linha e com 3 espaços entre eles.

3.24 O processo de encontrar o maior número (i.e., o máximo de um conjunto de números) é usado freqüentemente em aplicações computacionais. Por exemplo, um programa que determinasse o vencedor de um concurso de vendas receberia o número de unidades vendidas por vendedor. O vendedor que tivesse vendido mais unidades venceria o concurso. Escreva um pseudocódigo e depois um programa em C que receba uma série* de 10 números, determine o maioj deles e o imprima. Sugestão: Seu programa deve usar três variáveis da seguinte maneira:
contador: Um contador para contar até 10 (i.e., para controlar quantos números foram fornecidos e
para determinar quando todos os 10 números foram processados),
num: O número atual fornecido ao programa,
maior: O maior número encontrado em cada instante.

3.25 Escreva um programa em C que utilize um loop para imprimir a seguinte tabela de valores:

N	**10*N**	**100*N**	**1000*N**
1	**10**	**100**	**1000**
2	**20**	**200**	**2000**
3	**30**	**300**	**3000**
4	**40**	**400**	**4000**
5	**50**	**500**	**5000**
6 7 8 9	**60 70 80**	**600 700 800**	**6000 7000 8000 9000 10000**
10	**90 100**	**900 1000**	

O caractere de tabulação, **\t,** pode ser usado na instrução **printf** para separar as colunas por tabulações.

3.26 Escreva um programa em C que utilize um loop para produzir a seguinte tabela de valores:

3.27 Usando um método similar ao do Exercício 3.24, encontre os *dois* maiores valores de 10 números. Nota: Cada número só pode ser fornecido uma única vez.

A	**A+2**	**A+4**	**A+6**
3	**5**	**7**	**9**
6	**8**	**10**	**12**
9	**11**	**13**	**15**
12	**14**	**16**	**18**
15	**17**	**19**	**21**

3.28 Modifique o programa da Fig. 3.10 para validar suas entradas. Em qualquer entrada, se o valor fornecido for diferente de 1 ou 2, permaneça no loop até que o usuário forneça um valor válido.

3.29 O que o seguinte programa imprime?

```
#include <stdio.h>
main() {
int contador = 1;
while (contador <= 10) {
        printf("%s\n", contador % 2 ? "****" : "+ + + + + + + +");
         ++contador;
}
return 0;
}
```

3.30 O que o seguinte programa imprime?

```
#include <stdio.h>
main() {
int linha = 10, coluna;
while (linha >= 1) { coluna = 1;
        while (coluna <= 10) {
                printf ("?&s", linha % 2 ? "<" : ">"); ++coluna;
        }
        --linha; printf("\n");
}
return 0;
}
```

3.31 *(Problema de Else Oscilante)* Determine a saída de cada um dos programas seguintes quando **x** é 9 e **y** é 11 e quando xélleyé9. Observe que o compilador ignora os recuos em um programa em C. Além disso, o compilador C sempre associa o **else** com o **if** anterior, a menos que tenha sido dito algo em contrário pela colocação de chaves { }. Como, à primeira vista, o programador pode não ter certeza de a que **if** um **else** corresponde, isso é chamado problema do *"else* oscilante". Eliminamos os recuos dos códigos a seguir para tornar o problema mais interessante. (Sugestão: Aplique as convenções de recuos que você aprendeu.)

a) **if (x** < 10) **if (y** < 10)
printf("***\n");**
else

printf("#####\n");
printf("$$$$$\n"),•
b) **if (x** < 10) < **if (y** > 10)
printf("***\n");**
else {
printf("#####\n");
printf("$$$$$\n");
}

3.32 *(Outro Problema de Else Oscilante)* Modifique o código a seguir para produzir a saída mostrada. Use as técnicas apropriadas para os recuos. Você não pode fazer nenhuma modificação além de inserir chaves. O compilador ignora os recuos em um programa C. Eliminamos os recuos do código a seguir para tornar o programa mais interessante. Nota: E possível que não seja necessário fazer nenhuma modificação.

if (y == 8)
if (x == 5)
printf("@@@@@\n"); t a **else**
printf("#####\n"),• printf("$$$$$\n"); printf("&&&&&\n");

a) Admitindo **x =** 5ey = 8, a seguinte saída é produzida.

```
@@@@@
 $  $  $$  $
& & & & &
```

b) Admitindo x = 5 e y = 8, a seguinte saída é produzida.

```
@@@@@
```

c) Admitindo x = 5 e y =8, a seguinte saída é produzida.

```
@@@@@
&&&&&
```

d) Admitindo x = 5 e y = 8, a seguinte saída é produzida.Nota: As três últimas instruções **printf** são parte de uma instrução composta.

```
 # # # # # #
 $  $  $$  $
& & & & &
```

3.33 Escreva um programa que leia o lado de um quadrado e então imprima o quadrado com asteriscos. Seu programa deve funcionar com quadrados de todos os tamanhos entre 1 e 20. Por exemplo, se seu programa lesse um tamanho 4, deveria imprimir

```
****
****
****
****
```

3.34 Modifique o programa escrito no Exercício 3.33 de forma que seja impresso um quadrado vazado. Por exemplo, se seu programa lesse um tamanho 5, deveria imprimir

```
*****
*   *
*   *
*   *
*****
```

3.35 Um palíndromo é um número ou texto que é lido da mesma forma tanto da direita para a esquerda como da esquerda para a direita. Por exemplo, cada um dos inteiros seguintes, de cinco dígitos, é palíndromo: 12321,55555,45554 e 11611. Escreva um programa que leia um inteiro de cinco dígitos e determine se ele é palíndromo ou não. (Sugestão: Use os operadores divisão e resto, ou modulus, para separar o número em seus algarismos isolados.)

3.36 Obtenha um inteiro contendo apenas Os e ls (i.e., um inteiro "binário") e imprima seu valor equivalente na base dez. (Sugestão: Use os operadores resto e divisão para selecionar um a um os dígitos do número "binário", da direita para a esquerda. Da mesma forma que o sistema decimal de numeração onde o dígito da extremidade direita tem um valor posicionai 1 e o dígito imediatamente à sua esquerda tem um valor posicionai 10, e depois 100 e depois 1000 etc, no sistema binário o dígito da extremidade direita tem o valor posicionai 1, o dígito imediatamente à sua esquerda tem o valor posicionai 2, e depois 4, e depois 8 etc. Assim, o número 234 pode ser interpretado como 4*1+3*10 + 2*100. O equivalente decimal do binário 1101 é 1 * 1 + 0*2 + 1*4+1*8 ou 1+0 + 4 + 8 ou 13.)

3.37 Ouvimos sempre como os computadores são velozes. Como você pode determinar a rapidez com que seu equipamento realmente funciona? Escreva um programa com um loop **while** que conte de 1 a 3.000.000, de 1 em 1. Toda vez que a contagem atingir um múltiplo de 1.000.000, imprima este número na tela. Use seu relógio para cronometrar quanto tempo leva cada milhão de repetições do loop.

3.38 Escreva um programa que imprima 100 asteriscos, um de cada vez. A cada dez asteriscos, seu programa deve imprimir um caractere de nova linha. (Sugestão: Conte de 1 a 100. Use o operador resto para reconhecer cada vez que o contador atingir um múltiplo de dez.)

3.39 Escreva um programa em C que leia um inteiro, determine quantos dígitos são

iguais a 7 e imprima esta informação.

3.40 Escreva um programa que exiba o seguinte padrão quadriculado

```
* * * * * * *
 * * * * * * *
* * * * * * *
 * * * * * * *
* * * * * * *
 * * * * * * *
* * * * * * *
 * * * * * * *
```

Seu programa só pode usar três instruções **printf,** uma da forma
printf("* ");
uma da forma
printf("   ")
e uma da forma
printf("\n");

3.41 Escreva um programa que fique imprimindo múltiplos de 2, ou seja, 2,4, 8, 16, 32,64 etc. Seu loop não deve terminar (i.e., você deve criai' um loop infinito). O que acontece quando esse programa é executado?

3.42 Escreva um programa que leia o raio de um círculo (como um valor **float),** calcule seu diâmetro, seu perímetro (circunferência) e sua área e imprima estes valores. Use o valor 3.14.159 para pi**.**

3.43 O que há de errado com a seguinte instrução? Escreva-a novamente para realizar o que provavelmente o programador está tentando fazer.
printf *{"%d",*   ++(xx + y)) ;

3.44 Escreva um programa que leia três valores **float** diferentes de zero e depois determine e imprima se eles podem representar os lados de um triângulo.

3.45 Escreva um programa que leia três inteiros diferentes de zero e depois determine e imprima se eles podem ser os lados de um triângulo retângulo.

3.46 Uma empresa deseja transmitir dados através do telefone, mas existe a preocupação de que seus telefones possam estar grampeados. Todos os seus dados são transmitidos como inteiros de quatro dígitos. A empresa pediu a você que escrevesse um programa para criptografar os dados de forma que eles possam ser transmitidos com mais segurança. Seu programa deve ler um inteiro de quatro dígitos e criptografá-lo da seguinte maneira: Substitua cada dígito pelo resultado da expressão *(soma daquele dígito com* 7) *modulus 10* (ou seja, o resto da divisão por 10 do número obtido pela soma daquele dígito com 7). Depois, troque o primeiro dígito pelo terceiro e troque o segundo dígito com o quarto. A seguir, imprima o inteiro criptografado. Escreva um programa separado que receba um inteiro criptografado e decriptografe-o para que seja obtido o número original.

3.47 O fatorial de um inteiro não-negativo *n é* escrito *n*! (lido como "fatorial de *n")* e é definido da seguinte maneira:

n! = n-(n — l)(n —* 2)   1 (para valores de *n* maiores que ou iguais a 1)

e

n! = 1 (para *n* = 0).

Por exemplo, 5! = 5*4*3*2*1 que é 120.

a) Escreva um programa que leia um inteiro não-negativo e depois calcule e imprima seu fatorial.

b) Escreva um programa que calcule o valor da constante matemática *e* usando a fórmula:
e = 1 + 1/1! + 1/2!+ ...

c) Escreva um programa que calcule o valor de e^x usando a fórmula

$e^{x = 1 + x/1! + x*x/2! + (X*X*X)/3!}$

4

Controle do Programa

Objetivos

- Ser capaz de usar as estruturas de repetição for e do/while. * Entender a seleção múltipla usando a estrutura switch .
- Ser capaz de usar as instruções de controle do programa break e continue.
- Ser capaz de usar os operadores lógicos.

Quem pode controlar seu destino ?
Wiliam Shakespeare
Otelo

A chave usada sempre brilha.
Benjamin Franklin

O homem é um animal feito com ferramentas.
Benjamin Franklin

Inteligência... é a faculdade de fazer objetos artificiais, especialmente ferramentas para fazer ferramentas.
Henry Bergson

Sumário

4.1 Introdução

Neste momento, o leitor deve se sentir familiarizado com o processo de escrever programas simples, porém completos, em C. Neste capítulo, a repetição é examinada mais detalhadamente, e mais outras estruturas de repetição, especificamente as estruturas for e do/while, são apresentadas, assim como a estrutura switch. Analisamos a instrução break para abandonar imediata e rapidamente determinadas estruturas de controle e a estrutura continue para ignorar o restante do corpo de uma estrutura de repetição e passar para a próxima iteração do loop. O capítulo analisa os operadores lógicos usados para combinar condições e conclui com um resumo dos princípios de programação estruturada, apresentados nos Capítulos 3 e 4.

4.2 Os Fundamentos da Repetição

Muitos programas envolvem repetições ou *loops*. Um *loop* é um grupo de instruções que o computador executa repetidamente enquanto alguma *condição de continuação de loop* permanecer verdadeira. Vimos duas maneiras de realizar uma repetição:

1. *Repetição controlada por contador*
2. *Repetição controlada por sentinela*

Algumas vezes, a repetição controlada por contador é chamada *repetição definida* porque sabemos antecipadamente quantas vezes o loop será executado. A repetição controlada por sentinela é chamada algumas vezes *repetição indefinida* porque não se sabe antecipadamente quantas vezes o loop será executado.

Em uma repetição controlada por contador, uma *variável de controle* é usada para contar o número de repetições. A variável de controle é incrementada (normalmente de 1) cada vez que o grupo de instruções é realizado. Quando o valor da variável de controle indicar que o número correto de repetições foi realizado, o loop é encerrado e o computador continua a execução do programa a partir da instrução imediatamente após o loop.

Os valores sentinelas são usados para controlar repetições quando:

1. *O número exato de repetições não é conhecido de antemão e*
2. *O loop inclui instruções que obtêm dados cada vez que o loop é realizado.*

O valor sentinela indica o "fim dos dados". A sentinela é entrada depois que todos os itens regulares de dados tiverem sido fornecidos ao programa. As sentinelas devem ter valores diferentes daqueles dos itens regulares de dados.

4.3 Repetição Controlada por Contador

A repetição controlada por contador exige:

1. O *nome* de uma variável de controle (ou contador do loop).
2. O *valor inicial* da variável de controle.
3. O *incremento* (ou *decremento)* pelo qual a variável de controle é modificada cada vez que o loop é realizado.
4. A condição que testa o *valor final* da variável de controle (i.e., se o loop deve continuar).

Veja o programa simples mostrado na Fig. 4.1, que imprime os números de 1 a 10. A declaração

int contador = 1;

fornece o nome da variável de controle (contador), declara-a como sendo um inteiro, reserva espaço para ela e define seu *valor inicial* como 1. Esta declaração não é uma instrução executável.

A declaração e a inicialização de contador também poderiam ter sido realizadas com as instruções

int contador;
contador = 1;

A declaração não é executável, mas a atribuição é. Usamos ambos os métodos de inicializar variáveis. A instrução

++contador;

```
/* Repetição controlada por contador */
#include <stdio.h>
main(){
int contador = 1;                  /* inicialização */
while (contador <= 10)  {          /* condição de repetição */
        printf  ("%d\n", contador);
        ++contador;                        /* incremento */
}
return 0;
}
/* Repetição controlada por contador */
#include <stdio.h>
```

```
1
2
3
4
5
6
8
9
10
```

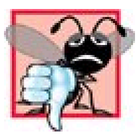
Fig. 4.1 Repetição controlada por contador.

incrementa o contador do loop de 1 cada vez que o loop é realizado. A condição de continuação do loop na estrutura while examina se o valor da variável de controle é menor que ou igual a 10 (o último valor para o qual a condição é verdadeira). Observe que o corpo do while é realizado mesmo quando a variável de controle é igual a 10. O loop termina quando a variável de controle se torna maior que 10 (i.e., quando contador se torna 11).

Normalmente, os programadores da linguagem C fariam o programa da Fig. 4.1 de uma forma mais concisa inicializando contador com o valor 0 e substituindo a estrutura while por

```
while ( ++contador <= 10)
        printf  ("%d\n", contador);
```

Este código economiza uma instrução porque o incremento é realizado diretamente na condição de while antes de ela ser examinada. Além disso, esse código elimina as chaves em torno do corpo do while porque agora o while contém apenas uma instrução. Fazer códigos de uma maneira condensada como essa exige alguma prática.



Erro comun de programação 4.1

Como os valores em ponto flutuante podem ser valores aproximados, controlar a contagem de loops com variáveis de ponto flutuante pode resultar em valores imprecisos de contadores e exames incorretos da condição de terminação.



Boa prática de programação 4.1

Controle a contagem das repetições (loops) com valores inteiros.



Boa prática de programação 4.2

Faça um recuo nas instruções do corpo de cada estrutura de controle.

Boa prática de programação 4.3

Coloque uma linha em branco antes e depois de cada uma das principais estruturas de controle para fazer com que ela se destaque no programa.

Boa prática de programação 4.4

Muitos níveis de aninhamento podem fazer com que um programa fique difícil de entender. Como regra geral, tente evitar o uso de mais de três níveis de recuos

Boa prática de programação 4.5

A combinação de espaçamento vertical antes e após as estruturas de controle e dos recuos do corpo daquelas estruturas dá aos programas um aspecto bidimensional que aumenta muito a sua legibilidade.

4.4 A Estrutura de Repetição For

A estrutura de repetição for manipula automaticamente todos os detalhes da repetição controlada por contador. Para ilustrar o poder do for, vamos escrever novamente o programa da Fig. 4.1. O resultado está mostrado na Fig. 4.2.

O programa funciona da seguinte maneira. Quando a estrutura for começa a ser executada, a variável de controle contador é inicializada com o valor 1. A seguir, a condição de continuação do loop contador <= 10 é examinada. Como o valor inicial de contador é 1, a condição é satisfeita, portanto a instrução printf imprime o valor de contador, ou seja, 1.

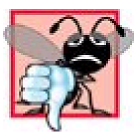
A variável de controle contador é então incrementada pela expressão contador++ e o loop começa novamente com seu teste de continuação. Como a variável de controle é agora igual a 2, o valor final não é excedido, e o programa realiza a instrução printf mais uma vez. O processo continua até que a variável de controle contador seja incrementada até o seu valor final de 11 — isto faz com que o teste de continuação do loop não seja verdadeiro e a repetição termine. O programa continua com a execução da primeira instrução depois da estrutura for (neste caso, a instrução return no final do programa).

```
/* Repetição controlada por contador com a estrutura for */
#include <stdio.h>
main()
{
int contador;
/* inicialização, condição de repetição e incremento */
/* estão incluidos no cabecalho da estrutura */
for  (contador = 1; contador <= 10; contador++)
        printf("%d\n", contador);
return 0;
}
```

Fig. 4.2 Repetição controlada por contador com a estrutura **for.**

A Fig. 4.3 examina mais detalhadamente a estrutura for da Fig. 4.2. Observe que a estrutura for "faz tudo" — ela especifica todos os itens necessários para a repetição controlada por contador com uma variável de controle. Se houver mais de uma instrução no corpo do for, devem ser usadas chaves para definir o corpo do loop.

Observe que a Fig. 4.2 usa a condição de continuação do loop contador <= 10. Se o programador escrever incorretamente contador < 10, o loop só seria executado 9 vezes. Esse é um erro comum de lógica chamado *off-by-one error (erro por falta de uma repetição).*



Erro comun de programação 4.2

Usar um operador relacional inadequado ou um valor final incorreto de um contador de loop na condição de uma estrutura ***while*** *ou* ***for*** *pode causar erros de falta de uma repetição (off-by-one).*



Boa prática de programação 4.6

Usar o valor final na condição de uma estrutura ***while*** *ou* ***for*** *e usar o operador relacionai <= evitará erros por falta de uma repetição (off-by-one). Para um loop usado para imprimir os valores 1 a 10, por exemplo, a condição de continuação do loop deve ser* ***contador <= 10*** *em vez de* ***contador < 11*** *ou* ***contador < 10.***

O formato geral da estrutura for é for *(expressão1; expressão2; expressão3) instrução*onde *expressão1* inicializa a variável de controle do loop, *expressão2 é* a condição de continuação do loop e *expressão3* incrementa a variável de controle. Na maioria dos casos, a estrutura for pode ser representada por uma estrutura while equivalente da seguinte maneira:

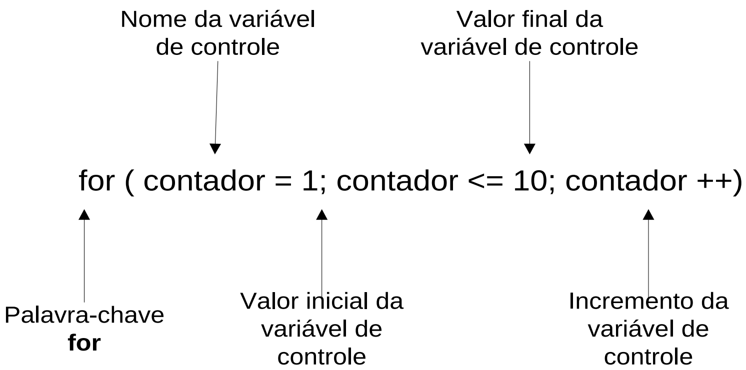


Fig. 4.3 Componentes de um cabecalho típico da estrutura for.

expressão1;
while ***(expressão2) {***
instrução expressão3;
}

Essa regra tem uma exceção, que será vista na Seção 4.9.

Freqüentemente, *expressão1* e *expressão3* são listas de expressões separadas por vírgulas. As vírgulas são usadas aqui na forma de *operadores de vírgulas* que garantem que a lista de expressões seja avaliada da esquerda para a direita. O valor e o tipo de uma lista de expressões que pode ser separada por vírgulas é o valor e o tipo da expressão mais à direita da lista. O operador de vírgula é usado mais freqüentemente em uma estrutura for. Sua utilidade principal é permitir que o programador use expressões múltiplas para inicialização e/ou incremento. Por exemplo, pode haver duas variáveis de controle em uma única instrução for que devem ser inicializadas e incrementadas.



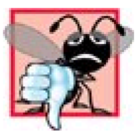
Boa prática de programação 4.7

Nas seções de inicialização e incremento de uma estrutura for, coloque apenas expressões que utilizem variáveis de controle. A manipulação de outras variáveis deve aparecer antes do loop (se elas devem ser executadas apenas uma vez como instruções de inicialização) ou no corpo do loop (se elas devem ser executadas uma vez. para cada repetição como as instruções para incrementar ou decrementar).

As três expressões na estrutura for são opcionais. Se *expressão2* for omitida, a linguagem C pressume que a condição é verdadeira, criando assim um loop infinito. A *expressão1* pode ser omitida se a variável de controle for inicializada em outro lugar do programa. A *expressão3* pode ser omitida se o incremento é calculado por instruções no corpo da estrutura for ou se nenhum incremento se faz necessário. A expressão de incremento na estrutura for age como uma instrução independente do C no final do corpo do for. Desta forma, as expressões

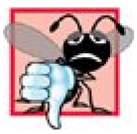
contador = contador + 1
contador += 1
++contador
contador++

são equivalentes na parte incrementai da estrutura for. Muitos programadores em C preferem a forma contador++ porque o incremento ocorre depois de o corpo do loop ser executado. Assim, pós-incrementar parece mais natural. Como a variável que está sendo pós-incrementada ou pré-incrementada aqui não aparece na expressão, ambas as formas de incrementar produzem o mesmo efeito. São exigidos os dois ponto-e-vírgulas na estrutura for.



Erro comun de programação 4.3

Usar vírgulas em vez de ponto-e-vírgula em um cabeçalho de uma estrutura ***for.***



Erro comun de programação 4.4

Colocar um ponto-e-vírgula imediatamente à direita do cabeçalho de uma estrutura ***for*** *faz com que o corpo dessa estrutura seja uma instrução vazia. Normalmente isto é um erro de lógica.*

4.5 A Estrutura For: Notas e Observações

1. A inicialização, a condição de continuação do loop e o incremento podem conter expressões aritméticas. Por exemplo, admita que x = 2 e y = 10, a instrução

   for (j = x; j <= 4 * x * y; j += y / x)

   é equivalente à instrução

   for (j = 2; j <= 80; j += 5)

2. O "incremento" pode ser negativo (caso em que na realidade ele é um decremento e o loop faz.. uma contagem regressiva).

3. Se a condição de continuação do loop for inicialmente falsa, a parte do corpo do loop não é realizada. Desta forma, a execução prossegue com a instrução imediatamente após a estrutura for.

4. Freqüentemente, a variável de controle é impressa ou usada em cálculos no corpo de um loop, mas não é exigido que isto aconteça. É comum usar a variável de controle para controlar a repetição sem nunca mencioná-la no corpo do loop.

5. A estrutura for é representada em um fluxograma da mesma forma que a estrutura while. Por exemplo, o fluxograma da instrução for

```
for (contador = 1; contador <= 10; contador+ + )
printf("%d", contador);
```

é mostrada na Fig. 4.4. Esse fluxograma torna claro que a inicialização ocorre apenas uma vez e que o incremento acontece depois de o corpo da instrução ter sido executado. Observe que (além dos pequenos círculos e setas) o fluxograma contém apenas símbolos retangulares e um símbolo na forma de um losango. Imagine, novamente, que o programador tem uma grande caixa de estruturas for vazias — tantas quanto ele possa precisar empilhar ou aninhar com outras estruturas de controle para formar uma implementação estruturada do fluxo de controle de um algoritmo. E os retângulos e losangos sejam preenchidos com ações e decisões apropriadas ao algoritmo.



Boa prática de programação 4.8

Embora o valor da variável de controle possa ser modificado no corpo de um loop for, isto pode levar a erros difíceis de perceber.

4.6 Exemplos Usando a Estrutura For

Os exemplos a seguir mostram métodos de mudar o valor da variável de controle em uma estrutura for.

a) Fazer a variável de controle assumir valores de 1 a 100 em incrementos de 1.

for (i = 1; i <= 100; i++)

b) Fazer a variável de controle assumir valores de 100 a 1 em incrementos de — 1 (decrementos de 1).

for (i = 100; i >= 100; i--)

c) Fazer a variável de controle assumir valores de 7 a 77 em passos de 7

for (i = 7; i <= 77; i += 7)

d) Fazer a variável de controle assumir valores de 2 0 a 2 em passos de -2.

for (i = 20; i >= 2; i -= 2)

e) Fazer a variável de controle assumir os valores da seguinte seqüência: 2, 5, 8, 11, 14, 17, 20.

for (j = 2; j <= 20; j += 3)

f) Fazer a variável de controle assumir os valores da seguinte seqüência: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

for (j = 99; j >= 0; j -= 11)

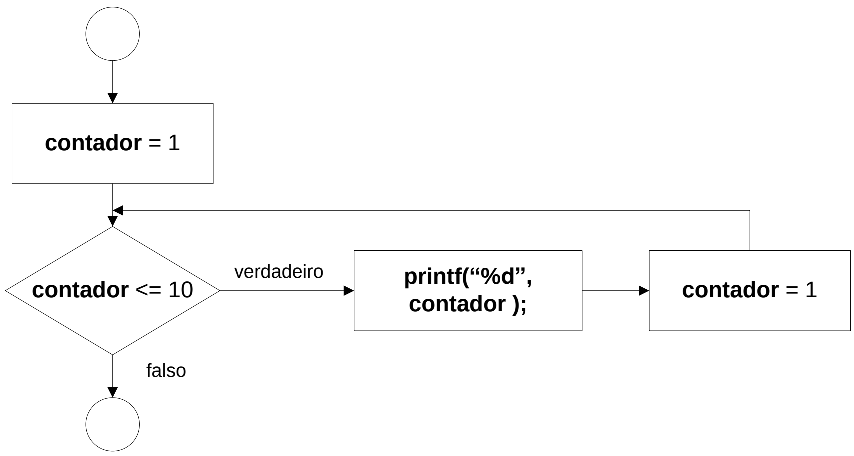


Fig. 4.4 Fluxograma de uma estrutura for típica.

Os dois exemplos a seguir fornecem aplicações simples da estrutura for. O programa da Fig. 4.5 usa a estrutura for para fazer a soma de todos os números inteiros pares de 2 a 100.

Observe que o corpo da estrutura for da Fig. 4.5 poderia ser incluído na parte mais à direita do cabeçalho do for usando o operador vírgula da maneira que se segue:

for (numero = 2; numero <= 100; soma += numero, numero += 2);

A inicialização soma = 0 também poderia ser incluída na seção de inicialização do for.



Boa prática de programação 4.9

Embora instruções que sejam anteriores à estrutura for ou que façam parte do corpo da estrutura possam ser incluídas freqüentemente no cabeçalho do for, evite fazer isto porque o programa pode ficar mais difícil de ler.



Boa prática de programação 4.10

Se possível, limite em uma linha o tamanho dos cabeçalhos das estruturas de controle.

```
/* Soma com for */
#include <stdio.h>
main( ) {
int soma = 0, numero;
for (numero = 2; numero <= 100; numero += 2)
soma += numero;
printf("A soma e %d\n", soma);
return 0;
}
```

A soma e 2550

Fig. 4.5 Soma com **for.**

O próximo exemplo calcula juros compostos usando a estrutura for. Imagine o seguinte enunciado de problema:

Uma pessoa investe $1000,00 em uma conta de poupança que rende juros de 5 por cento. Admitindo que todos os juros são deixados em depósito na conta, calcule e imprima a quantia na conta ao final de cada ano, ao longo de 10 anos. Use a seguinte fórmula para determinar estas quantias:

a = p(l + r)^n

onde:

p é a quantia investida originalmente (i.e., o valor principal) r é a taxa anual de juros

n é o número de anos (nota do escaniador, é na N, ou seja tudo o que tá no parenteses elevado na n)

a é a quantia existente em depósito no final do n-ésimo ano.

Esse problema envolve um loop que realiza os cálculos apropriados para cada um dos 10 anos durante os quais o dinheiro permanece depositado. A solução é mostrada na Fig. 4.6.

A estrutura for executa o corpo do loop 10 vezes, modificando o valor da variável de controle de 1 a 10 com incrementos de 1. Embora a linguagem C não inclua um operador exponencial, podemos, entretanto, usar a função pow da biblioteca padrão com esta finalidade. A função pow (x, y) calcula o valor de x elevado à potência y. Ela necessita de dois argumentos do tipo double e retorna um valor double. O tipo double é um tipo de ponto flutuante semelhante ao float, mas a variável do tipo double pode armazenar um valor muito maior e com maior precisão do que o float. Observe que, sempre que uma função matemática, como power, for usada, deve ser incluído o arquivo de cabeçalho math.h.

```
/* Calculando juros compostos */
#include <stdio.h>
#include <math.h>
main ()
{
int ano;
double quantia, principal = 1000.0,  taxa = .05;
printf  ("%4s%21s\n",   "Ano",   "Saldo na conta");
for  (ano = 1; ano <= 10; ano++) {
        quantia = principal * pow(1.0 = taxa, ano);
        printf("%4d%21.2f\n",  ano, quantia);
}
return 0;
}
```

Ano	**Saldo na conta**
1	**1050.00**
2	**1102.50**
3	**1157.62**
4	**1215.51**
5	**1276.28**
6	**1340.10**
7	**1407.10**
8	**1477.46**
9	**1551.33**
10	**1628.89**

Fig. 4.6 Calculando juros compostos com **for.**

Na verdade, esse programa não funcionaria corretamente sem a inclusão de **math.h.** A função **pow** exige dois argumentos **double.** Observe que **ano** é um inteiro. O arquivo **math.h** inclui informações que dizem ao compilador para converter o valor de **ano** para uma representação **double** antes de chamar a função. Estas informações estão contidas em algo chamado *protótipo de função* de **pow.** Os protótipos de funções são um recurso novo e importante do ANSI C e são explicados no Capítulo 5. Fornecemos um resumo da função **pow** e de outras funções matemáticas da biblioteca no Capítulo 5.

Observe que declaramos as variáveis **quantia, principal** e **taxa** com o tipo **double.** Fizemos isto visando a ter mais simplicidade porque estamos lidando com partes fracionárias de dólar.



Boa prática de programação 4.11

Não use variáveis do tipo float e double para realizar cálculos financeiros. A imprecisão dos números de ponto flutuante pode causar erros que resultarão em valores incorretos. Nos exercícios, exploramos o uso de inteiros para realizar cálculos financeiros.

Eis uma explicação simples do que pode dar errado quando **float** e **double** são usados para representar quantias de dinheiro.

Dois valores monetários do tipo **float** armazenados no equipamento poderiam ser 14.234 (que é impresso como 14.23 com %.**2f**) e 18.673 (que é impresso como 18.67). Quando essas quantias são adicionadas, é obtida a soma 32.907, que é impressa como 32.91. Assim, a saída de impressão poderia parecer como

14 .23 + 18.67
32.91

mas claramente a soma das parcelas impressas deveria ser 32.90! Tome cuidado!

O especificador de conversão *%2*1**.2f** é usado para imprimir o valor da variável **quantia** no programa. O **21** no especificador de conversão indica a *largura do campo* no qual o valor será impresso. **I** Uma largura de campo igual a **21** especifica que o valor impresso aparecerá em **21** posições de impressão. O **2** especifica a precisão (i.e., o número de posições decimais). Se o número de caracteres exibidos for menor que a largura do campo, o valor será automaticamente *alinhado (justificado) pela direita* no I campo. Isto é particularmente útil para alinhar valores de ponto flutuante com a mesma precisão. Para *alinhar (justificar) pela esquerda,* coloque um sinal - (subtração) entre o % e a largura do campo. Observe que o sinal de subtração também pode ser usado para justificar inteiros (como em %-**6d)** e strings de caracteres (como em **%-8s).**

Analisaremos com detalhes os poderosos recursos de formatação de **printf** e **scanf** no Capítulo 9.

4.7 A Estrutura de Seleção Múltipla Switch

No Capítulo 3, analisamos a estrutura de seleção simples **if** e a estrutura de seleção dupla **if/else.** Ocasionalmente, um algoritmo pode conter uma série de decisões nas quais uma variável ou expressão é testada separadamente para cada um dos valores constantes que ela pode assumir, e, com base nisso, diferentes ações são tomadas. A linguagem C fornece a estrutura de seleção múltipla para manipular tal tomada de decisão.

A estrutura **switch** consiste em uma série de rótulos **case** e de um caso opcional **default.** O programa da Fig. 4.7 usa **switch** para contar o número de letras de cada conceito (grau) diferente que os estudantes conseguiram no exame.

No programa, o usuário entra com as letras referentes aos conceitos (graus) de uma turma. Dentro do **cabeçalho** do **while,**

while ((grau = getchar ( )) != EOF)

a atribuição entre parênteses é executada em primeiro lugar. A função **getchar** (da biblioteca padrão de entrada/saída) lê um caractere do teclado e o armazena na variável inteira **grau.** Normalmente, os caracteres são armazenados em variáveis do tipo ***char.*** Entretanto, um recurso importante do C é que

```
/* Contando os conceitos */
#include <stdio.h>

main() {
int grau;
int aConta = 0, bConta = 0,  cConta = 0, dConta = 0,  fConta = 0;

printf("Entre com os conceitos.  \n");
printf("Entre com o caractere EOF (fim) para finalizar as entradas.\n");
while (  ( grau = getchar ()  )   != EOF) {
        switch (grau)  {          /* switch aninhado em um while */
                case  'A':  /* o grau foi A maiúsculo */
                case  'a':  /* ou a minúsculo */
                        ++aConta;
                        break;
                case  'B':  /* o grau foi B maiúsculo */
                case  'b':  /* ou b minúsculo */
                        ++bConta;
                        break;
                case 'C': /* o grau foi C maiúsculo */
                case 'c':  /* ou c minúsculo */
                        ++cConta;
                        break;
                case  'D':  /* o grau foi D maiúsculo */
                case  'd':  /* ou d minúsculo */
                        ++dConta;
                        break;
```

```
                case  'F':  /* o grau foi F maiúsculo */
                case  'f`:  /* ou f minúsculo */
                        ++fConta;
                        break;
                case  '\n': /* ignore isto na entrada */
                case  '   ':
                        break;
                default:          /* obtenha todos outros caracteres */
                        printf ("Fornecido um conceito incorreto.");
                        printf  (" Entre com um novo conceito.\n");
                        break;
}
}
printf("\nOs totais de cada conceito sao:\n");
printf("A: %d\n", aConta);
printf("B: %d\n", bConta);
printf("C: %d\n", cConta);
printf("D: %d\n", dConta);
printf("F: %d\n",  fConta);
return 0;
}
```

```
Entre com os conceitos.
Entre com o caractere EOF (fim) para finalizar as entradas.
A
B
C
C
A
D
F
C
E
Fornecido um conceito incorreto. Entre com um novo conceito.
D
A
B
Os totais de cada conceito sao:
A:      3
B:      2
C:      3
D:      2
F:      1
```

Fig. 4.7 Um exemplo usando **switch**.

os caracteres podem ser armazenados em qualquer tipo inteiro de dados porque são representados com inteiros de 1 byte no computador. Assim, podemos tratar um caractere tanto como um inteiro quanto como um caractere, dependendo do seu uso. Por exemplo, a instrução

printf("O caractere (%c) tem o valor %d.\n",   'a', 'a');

usa os especificadores de conversão ***%c*** e ***%d*** para imprimir o caractere a e seu valor inteiro, respectivamente. O resultado é

O caractere(a) tem o valor 97.

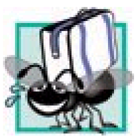
O inteiro 97 é a representação numérica do caractere no computador. Muitos computadores usam atualmente o *conjunto de caracteres ASCII (American Standard Code for Information Interchange)* no qual 97 representa a letra minúscula 'a'. Uma lista dos caracteres ASCII e seu valor decimal é apresentada no Apêndice D. Os caracteres podem ser lidos com scanf usando o especificador de conversão ***%c.***

Em geral, as instruções de atribuição possuem um valor. Este é exatamente o valor atribuído à variável no lado esquerdo do =. O valor da atribuição grau = getchar () é o caractere retornado por getchar e atribuído à variável grau.

O fato de que as instruções de atribuição possuem um valor pode ser útil para inicializar muitas variáveis com o mesmo valor. Por exemplo,

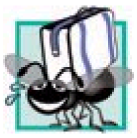
a = b = c = 0;

avalia primeiramente a atribuição c = 0 (porque o operador = faz associação da direita para a esquerda). A seguir, é atribuído à variável b o valor da atribuição c = 0 (que é 0). Então, é atribuído à variável a o valor da atribuição b = (c = 0) (que também é 0). No programa, o valor da atribuição grau = getchar () é comparado com o valor de EOF (um símbolo cujo acrônimo significa "end of file", ou "fim de arquivo"). Usamos EOF (que normalmente tem o valor — 1) como valor sentinela. O usuário digita uma combinação de teclas que depende de cada sistema para indicar o "fim do arquivo" ("end of file"), i.e., "Não tenho mais dados a fornecer". EOF é uma constante inteira simbólica definida no arquivo de cabeçalho <stdio.h> (veremos no Capítulo 6 como as constantes simbólicas são definidas). Se o valor atribuído a grau for igual a EOF, o programa é encerrado. Escolhemos representar caracteres nesse programa como ints porque **EOF** tem um valor inteiro (repetindo, normalmente — 1).



Dicas de portabilidade 4.1

A combinação de teclas para entrar com EOF (fim do arquivo, end of file) depende do sistema.



Dicas de portabilidade 4.2

Verificar a constante simbólica EOF em vez de -1 torna os programas mais portáteis. O padrão ANSI declara que EOF é um valor inteiro negativo (mas não necessariamente -1). Desta forma, EOF poderia ter diferentes valores em diferentes sistemas.

sistemas UNIX e muitos outros, o indicador **EOF** é fornecido digitando a seqüência

<return> <ctrl-d>

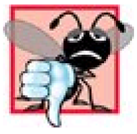
ta notação significa pressionar a tecla return e a seguir pressionar simultaneamente as teclas **ctrl** e **d.** Em outros sistemas como o VAX VMS, da Digital Equipment Corporation, ou o MS-DOS, da Microsoft Corporation, o indicador **EOF** pode ser fornecido digitando

<ctrl-z>

O usuário fornece os conceitos pelo teclado. Quando a tecla return (ou enter) é pressionada, os caracteres são lidos pela função **getchar,** um de cada vez. Se o caractere fornecido não for igual a **EOF.** a estrutura **switch** é empregada. A palavra-chave **switch** é seguida do nome da variável **grau** entre parênteses. Isso é chamado *expressão de controle.* O valor dessa expressão é comparado com cada **um** dos *rótulos (labels)* ***case.*** Suponha que o usuário forneceu a letra **C** como um conceito (grau). **C** é comparado automaticamente a cada **case** em **switch.** Se acontecer alguma igualdade **(case 'C'** :) as instruções para aquele **case** são executadas. No caso da letra **C, cConta** é incrementado de 1 e a estrutura **switch** é abandonada imediatamente com a instrução **break.**

A instrução **break** faz com que o controle do programa continue com a primeira instrução após a estrutura **switch.** A instrução **break** é usada porque, caso contrário, os **cases** em uma instrução **switch** seriam todos executados. Se **break** não fosse utilizado em nenhum local de uma estrutura **switch,** cada vez que houvesse uma igualdade na estrutura as instruções de todos os **cases** restantes seriam executadas. Esse recurso raramente é útil. Se não houver nenhuma igualdade, o caso **default** é executado e uma mensagem de erro é impressa.

Cada **case** pode ter uma ou mais ações. A estrutura **switch** é diferente de todas as outras estruturas, já que não são necessárias chaves em torno de ações múltiplas em um **case** de um **switch.** O fluxograma da estrutura geral de seleção múltipla **switch** (usar um **break** em cada **case)** é apresentado na Fig. 4.8. O fluxograma torna claro que cada instrução **break** no final de um **case** faz com que o controle saia imediatamente da estrutura **switch.** Mais uma vez, observe que (além dos pequenos círculos e 5) o fluxograma contém apenas retângulos e losangos. Imagine, novamente, que o programador tenha acesso a uma grande caixa de estruturas **switch** vazias — tantas quanto o programador possa precisar para empilhar ou aninhar com outras estruturas de controle para formar uma implementação estruturada do fluxo de controle de um algoritmo. E, novamente, os retângulos e losangos são então preenchidos com ações e decisões apropriadas ao algoritmo.



Erro comun de programação 4.5

Esquecer de colocar uma instrução ***break*** *necessária em uma instrução* ***switch.***



Boa prática de programação 4.12

Fornecer um caso ***default em*** *instruções* ***switch.*** *Os casos (ou* ***cases)*** *não testados explicitamente em uma estrutura switch são ignorados. O caso* ***defaul t*** *ajuda a evitar isto fazendo com que o programador adote um*

procedimento para processar condições excepcionais. Há situações nas quais não se tem necessidade do processamento ***default.***



Boa prática de programação 4.13

Embora as cláusulas ***case*** *e a cláusula do caso* ***default*** *em uma estrutura* ***switch*** *possam ocorrer em qualquer ordem, é considerado uma boa prática de programação colocar a cláusula* ***default*** *por último.*



Boa prática de programação 4.14

Em uma estrutura ***switch*** *em que a cláusula* ***default*** *está colocada por último, a instrução* ***break*** *não é exigida ali. Mas alguns programadores incluem este* ***break*** *para manter a clareza e a simetria com os outros* ***cases.***

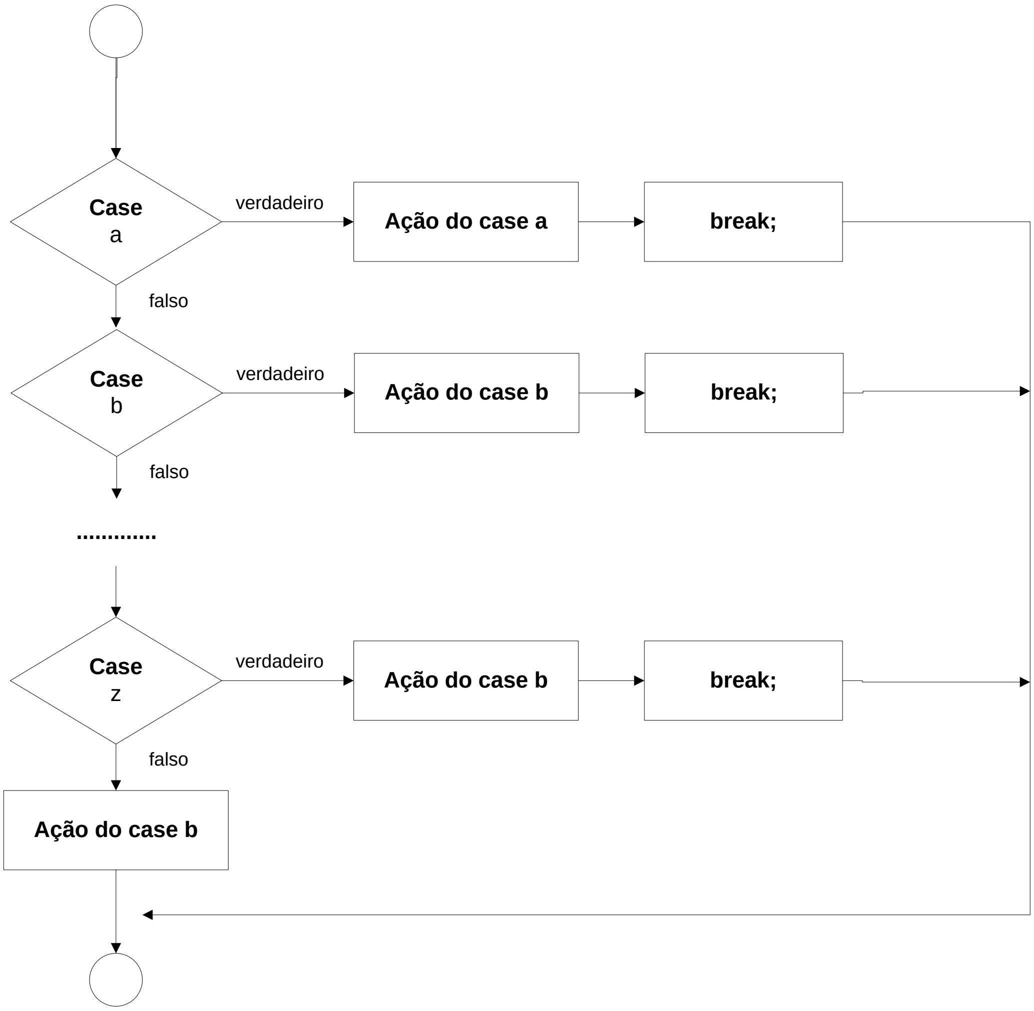


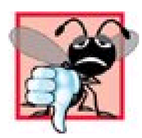
Fig. 4.8 A estrutura de seleção múltipla switch.

Na estrutura switch da Fig. 4.7, as linhas

```
case '\n':
case ' ':
break;
```

fazem com que o programa salte sobre os caracteres de nova linha em branco. Ler um caractere de cada vez pode causar alguns problemas. Para fazer com que o programa leia os caracteres, eles devem ser | enviados para o computador pressionando a *tecla return* no teclado. Isto faz com que o caractere de| nova linha seja colocado na entrada depois do caractere que desejamos processar. Freqüentemente, este j caractere de nova linha deve ser especialmente processado para que o programa funcione corretamente.

Incluindo os casos anteriores em nossa estrutura **switch,** evitamos que a mensagem de erro no caso **default** seja impressa todas as vezes em que um caractere de nova linha ou espaço for encontrado na entrada.



Erro comun de programação 4.6

Não processar caracteres de nova linha, na entrada de dados, ao ler um caractere de cada vez pode causar erros lógicos.



Boa prática de programação 4.15

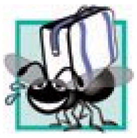
Lembre-se de fornecer recursos de processamento para os caracteres de nova linha na entrada de dados ao processar um caractere de cada vez.

Observe que vários rótulos (labels) de casos **(case)** listados em conjunto (como **case 'D': case 'd'** : na Fig. 4.7) significam simplesmente que o mesmo conjunto de ações deve ocorrer para qualquer um desses casos.

Ao usar a estrutura **switch,** lembre-se de que ela só pode ser usada para verificar uma *expressão constante inteira*, i.e., qualquer combinação de constantes de caracteres e constantes inteiras que levam a um valor inteiro constante. Uma constante de caractere é representada como um caractere específico entre aspas simples como '**A**'. Os caracteres devem estar entre aspas simples para serem reconhecidos somo constantes de caracteres. As constantes inteiras são simplesmente valores inteiros. Em nosso exemplo, usamos constantes de caracteres. Lembre-se de que os caracteres são realmente valores inteiros pequenos.

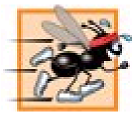
Linguagens portáteis como o C devem ter tipos de dados com tamanhos flexíveis. Aplicações diferentes podem precisar de inteiros de tamanhos diferentes. A linguagem C fornece vários tipos de dados para representar inteiros. O intervalo de valores inteiros para cada tipo depende do hardware específico de cada computador. Além dos tipos **int** e **char,** a linguagem C fornece os tipos **short** (uma abreviação de **short int)** e **long** (abreviação de **long int).** O padrão ANSI especifica que o intervalo de valores para os inteiros **short** é ±32767. Para a grande maioria dos cálculos inteiros, os inteiros **long** são suficientes. O padrão especifica que o intervalo mínimo de valores para os inteiros **long** é ±2147483647. Na maioria dos computadores, os **ints** são

equivalentes a **short** ou a **long.** O padrão declara que o intervalo de valores para um **int** é no mínimo o mesmo que o dos inteiros **short** e no máximo o dos inteiros **long.** O tipo de dados **char** pode ser usado para representar inteiros no intervalo ±127 ou qualquer um dos caracteres do conjunto de caracteres do computador.



Dicas de portabilidade 4.3

*Como os **ints** possuem tamanhos que variam de um sistema para outro, use inteiros **long** se for provável o processamento de inteiros além do intervalo ±32767 e se você quiser ser capaz de executar o programa em vários sistemas computacionais diferentes.*



Dica de desempenho 4.1

Em situações que possuem o desempenho como principal fator a ser considerado, onde a memória é valiosa ou a velocidade é necessária, pode ser desejável usar tamanhos menores de inteiros.

4.8 **A Estrutura de Repetição Do/While**

A estrutura de repetição **do/while** é similar à estrutura **while.** Na estrutura **while,** a condição de continuação do loop é testada em seu início antes de o corpo da estrutura ser executado. A estrutura **do/ while** testa a condição de continuação *depois* de o corpo do loop ser executado, portanto ele será executado pelo menos uma vez. Quando um **do/while** termina, a execução continua com a instrução após a cláusula **while.** Observe que não é necessário usar chaves na estrutura **do/while** se houver apenas uma instrução no corpo. Entretanto, normalmente as chaves são incluídas para evitar confusão entre as **Estruturas while** e **do/while.** Por exemplo,

while (*condição)* é considerado normalmente o cabeçalho para uma estrutura while. Uma estrutura do/while sem chaves em torno de um corpo com uma única instrução teria o seguinte aspecto

do
instrução
while *(condição);*

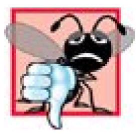
e poderia causar alguma confusão. A última linha — while *(condição)*; pode ser interpretada de maneira errada pelo leitor como uma estrutura while contendo uma instrução vazia. Desta forma, a instrução do/while com uma instrução é escrita freqüentemente como se segue para evitar confusão:

do {
instrução
} while *(condição) ;*



Boa prática de programação 4.16

*Alguns programadores sempre incluem chaves em uma estrutura **do/while** mesmo que elas não sejam necessárias. Isso ajuda a eliminar a ambigüidade entre a estrutura **do/while** contendo uma instrução e a estrutura **while.***



Erro comun de programação 4.7

*Acontecem loops infinitos quando a condição de continuação do loop em uma estrutura **while, for** ou **do/while** nunca se torna falsa. Para evitar isso, certifique-se de que não há um ponto-e-vírgula imediatamente após o cabeçalho de uma estrutura **while** ou **for.** Em um loop controlado por contador, certifique-se de que a variável de controle esteja sendo incrementada (ou decrementada) no corpo do loop. Em um loop controlado por sentinela, certifique-se de que o valor sentinela seja fornecido posteriormente.*

O programa da Fig. 4.9 usa uma estrutura do/while para imprimir os números de 1 a 10. Observe que a variável de controle contador é pré-incrementada no teste da condição de continuação do loop. Observe também o uso das chaves para encerrar um corpo com uma única instrução do do/while.

O fluxograma da estrutura do/while é apresentado na Fig. 4.10. Este fluxograma torna claro que a condição de continuação do loop não é executada até que a ação tenha sido realizada pelo menos uma vez. Mais uma vez, note que (além dos pequenos círculos e setas) o fluxograma contém apenas um retângulo e um losango. Imagine, novamente, que o programador tem acesso a uma grande caixa de estruturas do/while vazias — tantas quanto o programador possa precisar empilhar e aninhar com outras estruturas de controle para formar uma implementação estruturada do fluxo de controle de um algoritmo. E, mais uma vez, os retângulos e losangos são preenchidos com ações e decisões apropriadas ao algoritmo.

```
/* Usando a estrutura de repetição do/while */
#include <stdio.h>
main( ) {
int contador = 1;
do {
        printf("%d    ", contador);
}while   (++contador <= 10);
return 0;
}
```

1 2 3 4 5 6 7 8 9 10

Fig. 4.9 Usando a estrutura **do/while.**

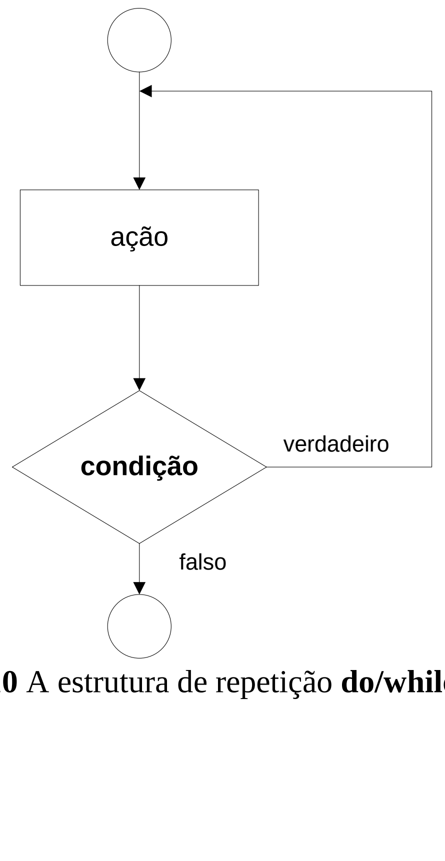


Fig. 4.10 A estrutura de repetição **do/while.**

4.9 As Instruções Break e Continue

As instruções break e continue são usadas para alterar o fluxo de controle. A instrução break, quando executada em uma estrutura while, for, do/while ou switch, faz com que aconteça a saida imediata daquela estrutura. A execução do programa continua com a primeira instrução depois da estrutura. Os usos comuns da instrução break são para sair prematuramente de um loop, ou para saltar sobre o restante de uma estrutura switch (como na Fig. 4.7). A Fig. 4.11 demonstra a instrução break em uma estrutura de repetição for. Quando a estrutura if detecta que x se tornou 5, o break é executado. Isto encerra a estrutura for e o programa continua com o printf após o for. O loop só é executado completamente quatro vezes.

```
/* Usando a instrução break em uma estrutura for */
#include <stdio.h>
main( ) {
int x;
for  (x = 1;  x <= 10;  x++) {
        if  (x == 5)
                break;  /* sai do loop somente se x == 5 */
        printf  ("%d ", x);
}
printf  ("\n Saiu do loop em x == %d\n", x); return 0;
}
```

```
 1 2 3 4
Saiu do loop em x == 5
```

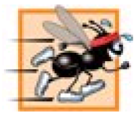
Fig. 4.11 Usando a instrução break em uma estrutura for.

A instrução continue, quando executada em uma estrutura while, for ou do/while, ignora (salta sobre) as instruções restantes no corpo daquela estrutura e realiza a próxima iteração do loop. *Em* estruturas while e do/while, o teste de continuação do loop é realizado imediatamente após a instrução continue ser executada. Na estrutura for, a expressão de incremento é executada e depois o | teste de continuação do loop é realizado. Anteriormente, afirmamos que a estrutura while poderia ser usada na maioria dos casos para representar a estrutura for. A única exceção ocorre quando a expressão de incremento na estrutura while vem após a instrução continue. Nesse caso, o incremento não é executado antes de a condição de continuação do loop ser testada, e o while não é executado da mesma maneira que o for. A Fig. 4.12 usa a instrução continue em uma estrutura for para ignorar a instrução printf na estrutura e começar a próxima iteração do loop.



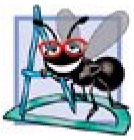
Boa prática de programação 4.17

Alguns programadores acham que break e continue violam as normas da programação estruturada. Como os efeitos dessas instruções podem ser conseguidos pelas técnicas de programação estruturada que aprenderemos em breve, esses programadores não usam break e continue.



Dica de desempenho 4.2

As instruções break e continue, quando usadas adequadamente, são executadas mais rapidamente que as técnicas estruturadas correspondentes que aprenderemos em breve.



Observação de engenharia de software 4.1

Há um conflito entre conseguir engenharia de software de qualidade e conseguir o software de melhor desempenho. Freqüentemente, um desses objetivos é atingido à custa do outro.

4.10 Operadores Lógicos

Até agora estudamos apenas condições simples como contador <= 10, total > 1000 e numero != valorSentinela. Expressamos essas condições em termos dos operadores relacionais >.<. >= e <=, e dos operadores de igualdade == e != . Cada decisão examinou exatamente uma condição. Se quiséssemos testar várias condições durante o processo de tomar uma decisão, tínhamos de realizar essa verificação em instruções separadas ou em estruturas if ou if/else aninhadas.

```
/* Usando a instrução continue em uma estrutura for */
#include <studio.h>
main( ) {
int x;
for  (x = 1; x <= 10;  x++) {
        if   (x == 5)
                continue;      /* ignora o código restante se x == 5 */
        printf{"%d  ",  x);
}
printf("Continue usado para ignorar a impressão do valor 5\n");
return 0;
}
```

```
1  2  3  4  6  7  8  9  10

Continue usado para ignorar a impressão do valor 5
```

Fig. 4.12 Usando a instrução **continue** em uma estrutura **for.**

A linguagem C fornece *operadores lógicos* que podem ser usados na criação de condições mais . complexas, combinando condições simples. Os operadores lógicos são *&& (E lógico),* I | *(OU lógico)* e !*(NÃO lógico* também chamado *negação lógica).* Veremos exemplos de cada um deles.

Suponha que, em algum ponto do programa, desejamos assegurar que duas condições sejam *verdadeiras* antes de escolher determinado caminho de execução. Nesse caso podemos usar o operador && como se segue:

if (sexo == 1 && idade >= 65)
++mulheresAposentadas ;

Essa instrução if contém duas condições simples. A condição sexo == 1 poderia ser utilizada, por exemplo, para determinar se a pessoa é do sexo feminino. A condição idade >= 65 é utilizada para determinar se a pessoa é um cidadão aposentado. As duas condições simples são avaliadas em primeiro lugar porque as precedências de == e >= são maiores que a precedência de &&. A instrução if verifica então a condição combinada

sexo == 1 && idade >= 65

Essa condição é verdadeira se e somente se ambas as condições simples forem verdadeiras. Finalmente, se realmente essa condição combinada for verdadeira, a contagem de mulheresAposentadas é incrementada de 1. Se ambas as condições ou uma delas for falsa, o programa ignora o processo de incrementar e prossegue com a instrução após o if.

A tabela da Fig. 4.13 resume o operador &&. A tabela mostra todas as quatro combinações possíveis de valores zero (falso) e diferente de zero (verdadeiro) para expressãol e expressão2. Freqüentemente, tais tabelas são conhecidas como *tabelas de verdade.* A linguagem C avalia todas as expressões que contêm operadores relacionais, operadores de igualdade e/ou operadores lógicos como 0 ou 1. Embora *a* linguagem C defina 1 como um valor verdadeiro, ela aceita *qualquer* valor diferente de zero como verdadeiro.

Agora vamos examinar o operador | I (OU lógico). Suponha que quiséssemos nos assegurar de que, em algum ponto do programa, uma de duas condições ou ambas sejam verdadeiras antes de escolher um determinado caminho de execução. Neste caso, usamos o operador I I como no seguinte segmento de programa:

```
if  (mediaSemestre >= 90  II  exameFinal >= 90)
        printf("O conceito do aluno e A\n");
```

Essa instrução também contém duas condições simples. A condição mediaSemestre >= 90 é utilizada para determinar se o aluno merece um "A" no curso devido a seu ótimo desempenho no semestre. A condição exameFinal >= 90 é utilizada para determinar se o aluno merece um "A" no curso devido ao seu excelente desempenho no exame final. A instrução if avalia então a condição combinada

mediaSemestre >= 90   II exameFinal >= 90

expressão1	expressão2	expressão 1 && expressão2
0	0	0
0	Diferente de zero	0
Diferente de zero	0	0
Diferente de zero	Diferente de zero	1

Fig. 4.13 Tabela de verdade para o operador && (E lógico).

expressão1	expressão2	expressão 1 | | expressão2
0	0	0
0	Diferente de zero	0
Diferente de zero	0	0
Diferente de zero	Diferente de zero	1

Fig. 4.14 Tabela de verdade para o operador 11 (OU lógico).

expressão	!expressão
0	**1**
Diferente de zero	**0**

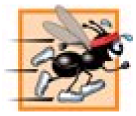
Fig. 4.15 Tabela de verdade para o operador! (negação lógica).

e concede um "A" ao aluno se uma das condições ou ambas forem verdadeiras. Observe que a mensagem "O conceito do aluno e A" só não é impressa quando ambas as condições simples forem falsas (zero). A Fig. 4.14 é uma tabela de verdade para o operador lógico OU (II).

O operador && tem uma precedência maior do que o | | . Ambos os operadores fazem associação da esquerda para a direita. Uma expressão contendo os operadores && ou I I é avaliada somente até sua f veracidade ou falsidade ser conhecida. Desta forma, a condição

```
sexo == 1 && idade >= 65
```

será encerrada se sexo não for igual a 1 (i.e., toda a expressão é falsa) e continuará se sexo for igual I a 1 (i.e., toda a expressão poderia ser verdadeira se idade >= 65).



Dica de desempenho 4.3

Em expressões que utilizam o operador &&, coloque na extremidade esquerda a condição que tem maior probabilidade de ser falsa. Em expressões que utilizam o operador I I, coloque na extremidade esquerda a condição com maior probabilidade de ser verdadeira. Isso pode reduzir o tempo de execução do programa.

A linguagem C fornece ! (negação lógica) para permitir que o programa "reverta" o significado de j uma condição. Diferentemente dos operadores && e I I, que combinam duas condições (e portanto são 1 operadores binários), o operador de negação lógica tem apenas uma condição como operando (e portanto é um operador unário). O operador de negação lógica é colocado antes de uma condição quando es- 1 tamos interessados em escolher um caminho de execução se a condição original (sem o operador de | negação lógica) for falsa, como no seguinte segmento de programa:

```
if  (!(conceito == valorSentinela))
        printf("O próximo conceito e %f\n", conceito);
```

Os parênteses em torno da condição grau == valorSentinela são necessários porque o operador de negação lógica tem uma precedência maior que o operador de igualdade. A Fig. 4.15 mostra a **1** tabela de verdade do operador de negação lógica.

Na maioria dos casos, o programador pode evitar o uso de uma negação lógica exprimindo diferentemente a condição com um operador relacionai adequado. Por exemplo, a instrução do exemplo anterior também pode ser escrita assim

```
if  (conceito != valorSentinela)
printf("O próximo conceito e %f\n", conceito);
```

O gráfico da Fig. 4.16 mostra a precedência e associatividade dos vários operadores C apresentados até aqui. Os operadores são mostrados de cima para baixo na ordem decrescente de precedência.

4.11 Confusão entre os Operadores de Igualdade (==) e Atribuição (=)

Há um tipo de erro que os programadores em C, seja qual for sua experiência anterior, tendem a cometer j com tanta freqüência que achamos que vale a pena tratar dele em uma seção separada. Esse erro é trocar acidentalmente os operadores de igualdade (==) e atribuição (=). O que torna essa troca tão prejudicial é fato de que elas normalmente não causam erros de sintaxe. Em vez disso, em geral as instruções com esses erros são compiladas corretamente e os programas são executados até o fim, gerando provavelmente resultados incorretos através de erros de lógica em tempo de execução.

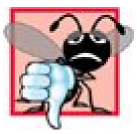
Há dois aspectos do C que causam esses problemas. Um é que qualquer expressão em C que produza um valor pode ser usado na parte de decisão de qualquer estrutura de controle. Se o valor for 0, ele é ido como falso, e, se for diferente de zero, é tratado como verdadeiro. O segundo é que as atribuições em C produzem um valor, ou seja, o valor que é atribuído à variável no lado esquerdo do operador de atribuição. Por exemplo, suponha que pretendemos escrever

```
if (codigoPagamento == 4)
        printf("Voce ganhou um bônus!");
```

mas que, acidentalmente, escrevemos

```
if  (codigoPagamento = 4)
        printf("Voce ganhou um bonus!");
```

A primeira instrução if concede adequadamente um bônus à pessoa cujo código de pagamento for igual a 4. A segunda instrução if — a que está errada — avalia a expressão de atribuição na condição if. Essa expressão é uma atribuição simples cujo valor é a constante 4. Como um valor diferente de zero é interpretado como "verdadeiro", a condição nessa instrução if é sempre verdadeira, e as pessoas sempre recebem um bônus, independentemente de quais sejam seus códigos de pagamento!



Erro comun de programação 4.8

Usar o operador == para atribuição ou usar o operador = para igualdade.

Operadores	Associatividade	Tipo
( )	Esquerda para a direita	Parênteses
++ -- + - ! (tipo)	Direita para esquerda	Unário
* ? %	Esquerda para a direita	Multiplicativo
+ -	Esquerda para a direita	Aditivo
< <= > >=	Esquerda para direita	Relacional
== !=	Esquerda para direita	Igualdade
&&	Esquerda para direita	E lógico
| |	Esquerda para direita	OU lógico
?:	Direita para esquerda	Condicional
= += -= *= /= %=	Direita para esquerda	Atribuição
,	Esquerda para a direita	Vírgula

Fig. 4.16 Precedência e associatividade dos operadores.

Normalmente, os programadores escrevem condições como x == 7 com o nome da variável à esquerda e a constante à direita. Invertendo isso, de forma que a constante esteja à esquerda e o nome da variável à direita, como em 7 == x, o programador que substituir acidentalmente o operador == pelo = estará protegido pelo compilador. O compilador tratará isso como um erro de sintaxe porque só variáveis podem ser colocadas no lado esquerdo de uma instrução de atribuição. Pelo menos isso evitará a destruição potencial de um erro lógico em tempo de execução.

Diz-se que os nomes de variáveis são l*values* (que significa "left values", ou "valores esquerdos") **1** porque podem ser usados no lado esquerdo de um operador de atribuição. As constantes são chamadas | *rvalues* (que significa "right values", ou "valores direitos") porque podem ser usadas apenas no lado **1** direito de um operador de atribuição. Observe que os valores esquerdos também podem ser usados como | valores direitos, mas o inverso não acontece.



Boa prática de programação 4.18

*Quando uma expressão de igualdade tem uma variável e uma constante, como em **x == 1,** alguns programadores preferem escrever a expressão com a constante à esquerda e o nome da variável à direita como proteção contra o erro lógico que ocorre quando o programador substitui acidentalmente o operador == .*

O outro lado da moeda também pode ser igualmente desagradável. Suponha que o programador deseja atribuir um valor a uma variável através de uma instrução simples como mas em vez disso escreve

Aqui, também, não há um erro de sintaxe. Em vez disso, o compilador simplesmente avalia a expressão | condicional. Se **x** for igual a **1,** a condição é verdadeira e a expressão retorna o valor 1. Se **x** não for **1** igual a **1,** a condição é falsa e a expressão retorna o valor 0. Seja qual for o valor retornado, não há operador de atribuição, portanto o valor é simplesmente perdido e o valor de **x** permanece inalterado. § causando provavelmente um erro lógico de tempo de execução. Infelizmente, não temos um procedimento prático disponível para ajudar você nesse problema!

4.12 Resumo de Programação Estruturada

Os programadores devem desenvolver seus programas da mesma forma que os arquitetos criam edifícios empregando o conhecimento comum de sua profissão. Nosso campo é mais recente do que a arquitetura, e nosso conhecimento é bem mais disperso. Aprendemos muita coisa em apenas cinco décadas. Talvez o mais importante seja que aprendemos que a programação estruturada produz programas mais fáceis de entender (do que a programação não-estruturada) e, portanto, mais fáceis de testar, depurar, modificar e até de se provar sua correção do ponto de vista matemático.

Os Capítulos 3 e 4 se dedicaram às estruturas de controle do C. Cada uma das estruturas foi apresentada, descrita em um fluxograma e analisada separadamente com exemplo. Agora, resumimos os resultados dos Capítulos 3 e 4 e apresentamos um conjunto simples de regras de formação e as propriedades dos programas estruturados.

A Fig. 4.17 resume as estruturas de controle analisadas nos Capítulos 3 e 4. Na figura, são usados pequenos círculos para indicar os únicos pontos de entrada e saída de cada estrutura. Conectar arbitrariamente símbolos isolados do fluxograma pode levar a programas não-estruturados. Assim, os programadores decidiram combinar símbolos de fluxogramas para formar um conjunto limitado de estruturas de controle e construir programas estruturados combinando adequadamente estruturas de controle de apenas duas maneiras. Para simplificar, são usadas somente estruturas de única entrada/única saída — há apenas um caminho para entrar e um caminho para sair de cada estrutura de controle. Conectar estruturas de controle em seqüência para formar programas estruturados é simples — o ponto de saída de uma estrutura de controle é conectado diretamente ao ponto de entrada da próxima estrutura de controle, i.e., as estruturas de controle são simplesmente colocadas uma após a outra em um programa; isso é chamado "empilhamento de estruturas de controle". As regras para a formação de programas estruturados também permitem que as estruturas de controle sejam aninhadas.

A Fig. 4.18 mostra as regras para criar adequadamente programas estruturados. As regras admitem \ que o símbolo retangular pode ser usado para indicar qualquer ação, incluindo entrada/saída (input/output).

Aplicar as regras da Fig. 4.18 sempre resulta em um fluxograma estruturado com um aspecto organizado de blocos de construção. Por exemplo, aplicar repetidamente a regra 2 ao fluxograma mais simples resulta em um fluxograma estruturado contendo muitos retângulos em seqüência (Fig. 4.20). Observe que a regra 2 gera uma pilha de estruturas de controle; portanto, vamos chamar a regra 2 de *regra de empilhamento.*

A regra 3 é chamada *regra de aninhamento.* Aplicar repetidamente a regra 3 ao fluxograma mais simples resulta em um fluxograma com estruturas de controle aninhadas organizadamente. Por exemplo, na Fig. 4.21, o retângulo no fluxograma mais simples é substituído inicialmente por uma estrutura de seleção dupla. A seguir, a regra 3 é aplicada novamente a ambos os retângulos na estrutura de dupla seleção, substituindo também cada um destes retângulos por estruturas de dupla seleção. Os quadros com linhas tracejadas em torno de cada uma das estruturas de dupla seleção representam o retângulo que foi substituído.

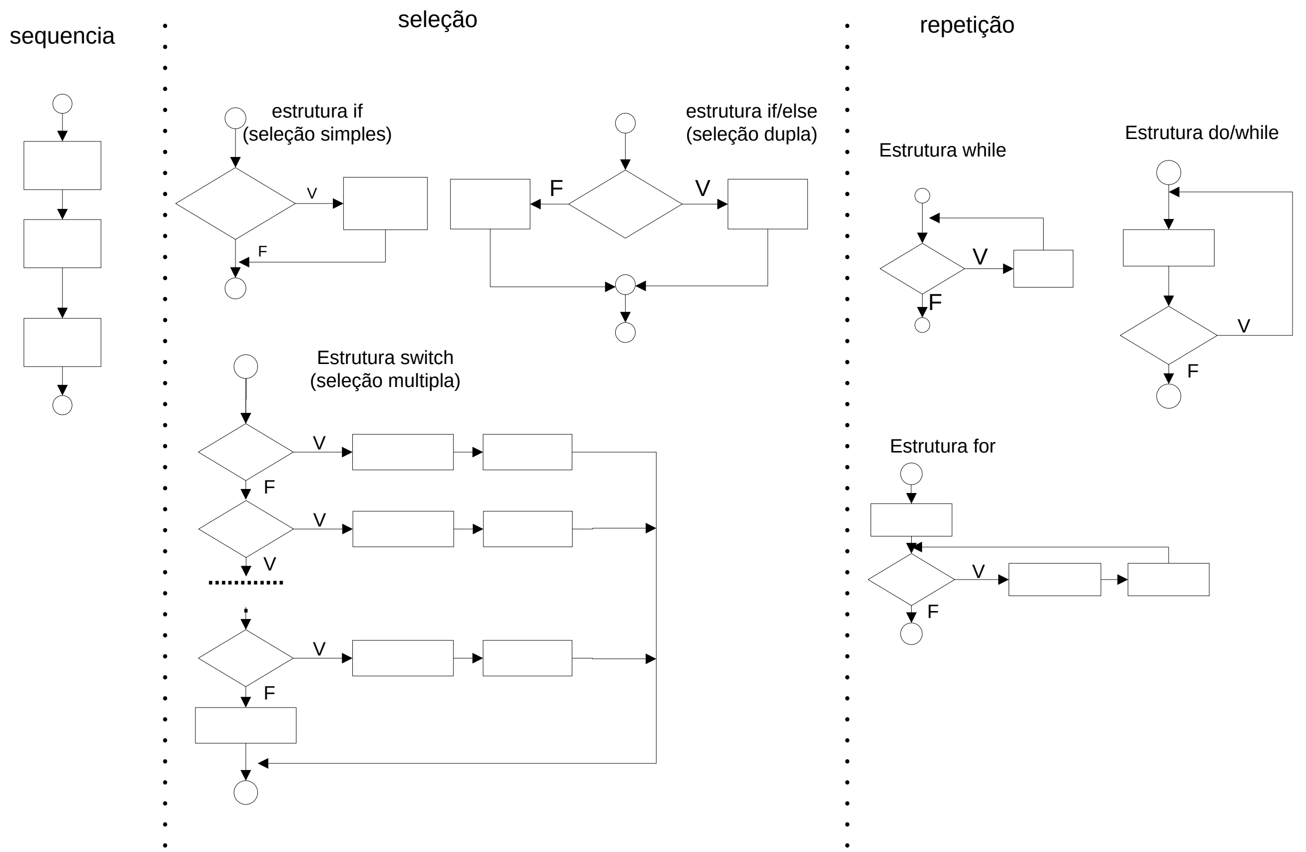


Fig. 4.17 Estruturas de seqüência, seleção e repetição, com única entrada/única saída, da linguagem C.

Regras para criação de programas estruturados

1) comece com o "fluxograma mais simples" (fig. 4.19)
2) Qualquer retângulo (ação) pode ser substituído por dois retângulos em sequencia.
3) Qualquer retângulo (ação) pode ser substituído por qualquer estrutura de controle (sequencia, IF, if/else, switch. While, do/while, ou for;
4) As regras 2 e 3 podem ser aplicadas com qualquer freqüência e em qualquer ordem.

Fig. 4.18 Regras para a criação de programas estruturados.

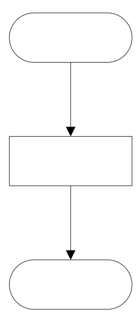


Fig. 4.19 O fluxograma mais simples.

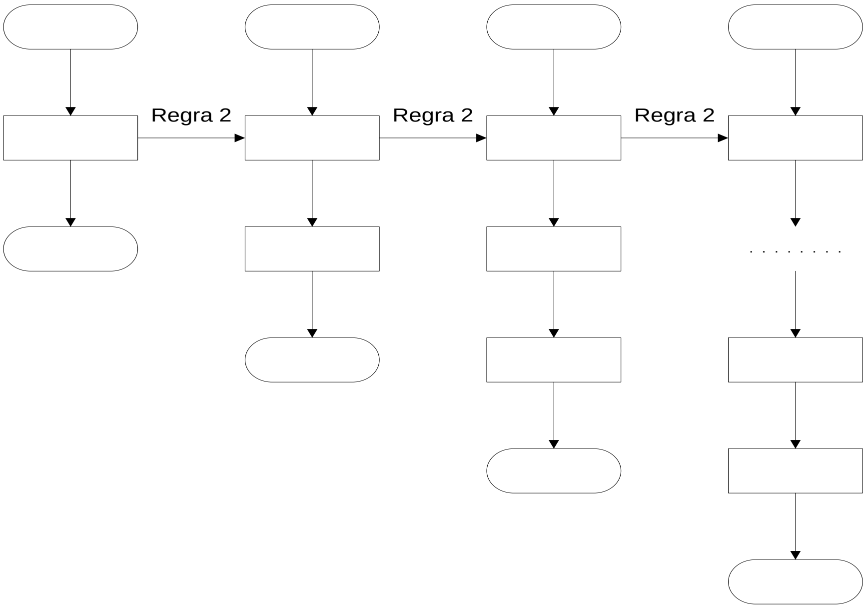


Fig. 4.20 Aplicando repetidamente a regra 2 da Fig. 4.18 ao fluxograma mais simples.

A regra 4 gera estruturas aninhadas maiores, mais envolventes e mais complexas. Os fluxogramas resultantes da aplicação das regras da Fig. 4.18 constituem o conjunto de todos os fluxogramas estruturados possíveis e portanto o conjunto de todos os programas estruturados possíveis.

Deve-se à eliminação da instrução goto o fato de esses blocos de construção nunca se sobreporem. A beleza do método estruturado está em usarmos apenas um pequeno número de peças simples de única entrada/única saída e as colocarmos de apenas duas maneiras. A Fig. 4.22 mostra os tipos de blocos de construção empilhados que surgem da aplicação da regra 2 e os tipos de blocos de construção aninhados que surgem da aplicação da regra 3. A figura também mostra o tipo de blocos de construção sobrepostos que não pode aparecer em fluxogramas estruturados (devido à eliminação da instrução goto).

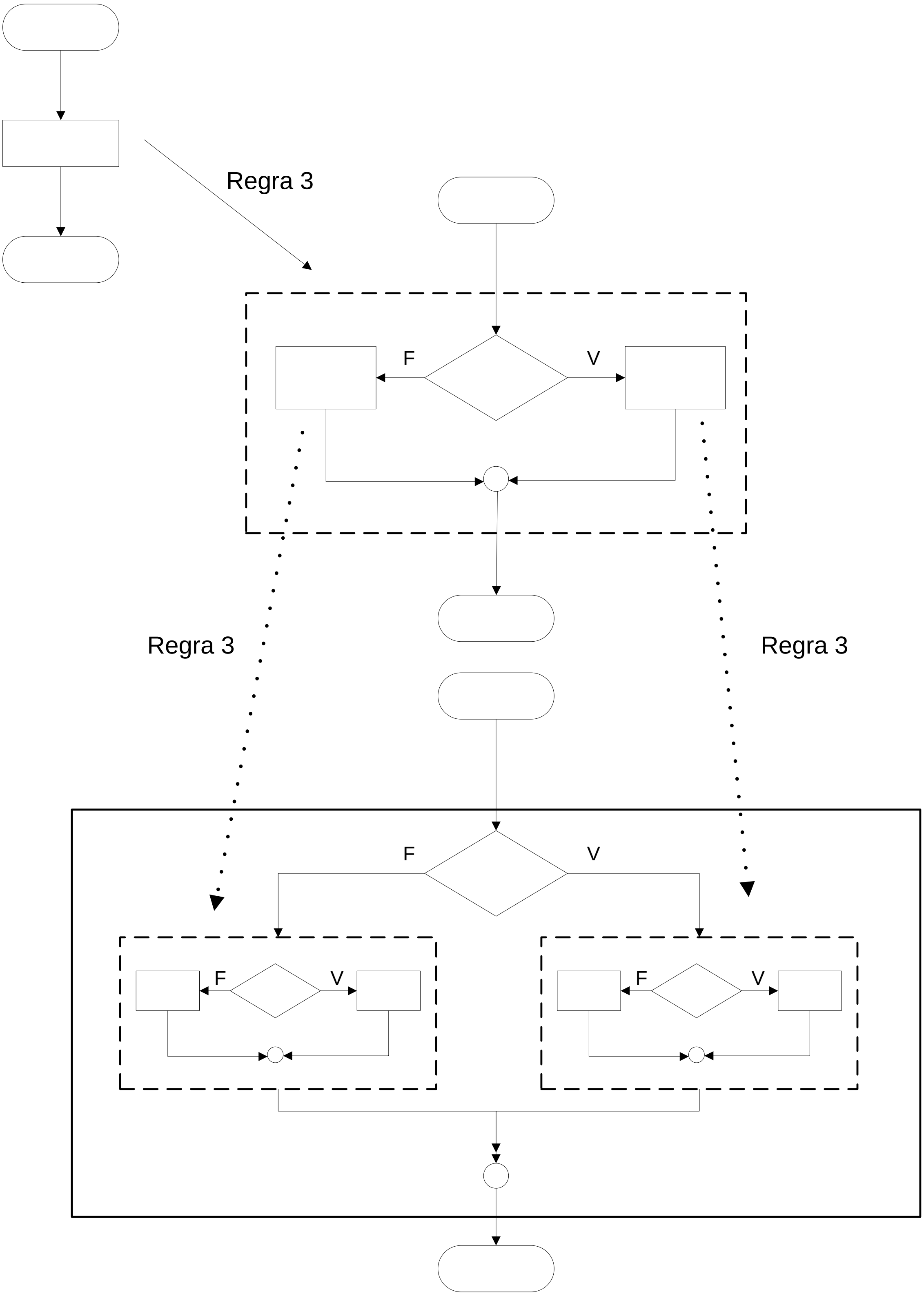


Fig. 4.21 Aplicando a regra 3 da Fig. 4.18 ao fluxograma mais simples.

Se as regras da Fig. 4.18 forem seguidas, um fluxograma estruturado (como o da Fig. 4.23) não poderá ser criado. Se você não tem certeza se um determinado fluxograma é estruturado, aplique as regras da Fig. 4.18 no sentido inverso para tentar reduzi-lo ao fluxograma mais simples. Se o fluxograma original puder ser reduzido ao fluxograma mais simples, ele é estruturado; caso contrário, não é.

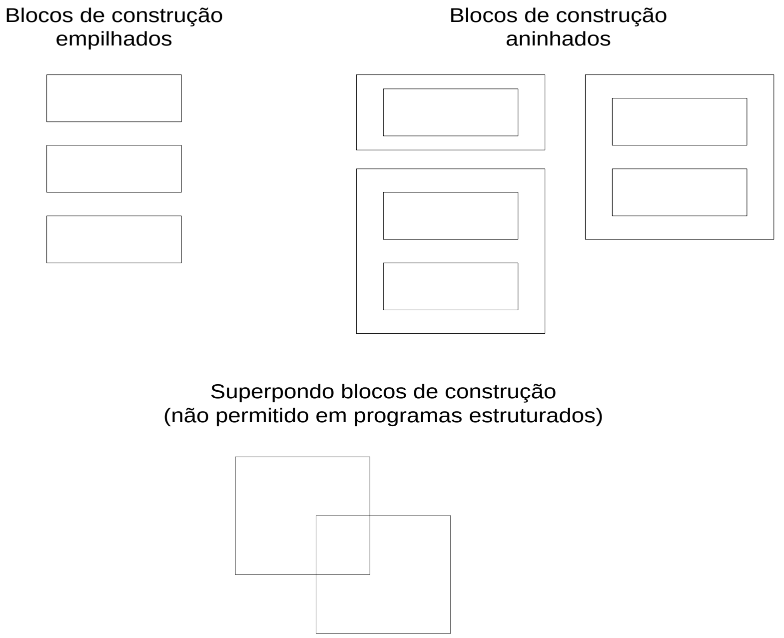


Fig. 4.22 Blocos de construção empilhados, aninhados e superpostos.

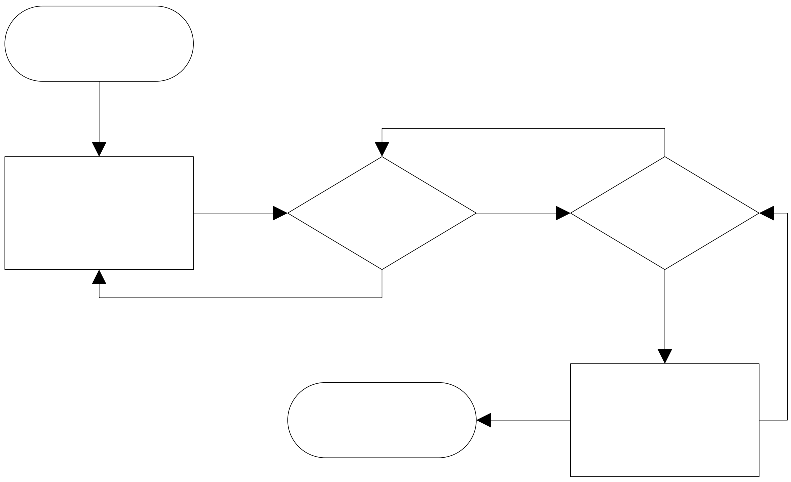


Fig. 4.23 Um fluxograma não-estruturado.

A programação estruturada favorece a simplicidade. Bohm e Jacopini nos fornecem a conclusão de que são necessárias apenas três formas de controle:

- Seqüência
- Seleção
- Repetição

A seqüência é trivial. A seleção é implementada de uma de três maneiras:

- estrutura **if**
- estrutura **if/else** (seleção dupla)
- estrutura **switch** (seleção múltipla)

realidade, é fácil provar que apenas a estrutura **if** é suficiente para fornecer qualquer forma de seleção — tudo que pode ser feito com as estruturas **if/else** e **switch** pode ser implementado com a estrutura **if.**

A repetição é implementada de uma entre três maneiras:

• estrutura **while**
• estrutura **do/while** • estrutura **for**

É fácil provar que a estrutura **while** é suficiente para fornecer qualquer forma de repetição. Tudo que pode ser feito com as estruturas **do/while** e **for** pode ser feito com a estrutura **while.**

Combinar esses resultados ilustra que qualquer forma de controle que venha a ser necessário em um programa em C pode ser expresso em termos de apenas três formas de controle:

- seqüência
- estrutura **if** (seleção)
- estrutura **while** (repetição)

E essas estruturas de controle podem ser combinadas de apenas duas maneiras - empilhamento e aninhamento. Como se pode ver, a programação estruturada realmente favorece a simplicidade.

Nos Capítulos 3 e 4, analisamos como criar programas a partir de estruturas de controle contendo ações e decisões. No Capítulo 5, apresentamos outra unidade de estruturação de programas chamada *função.* Aprenderemos a compor programas extensos combinando funções que, por sua vez, são compostas de estruturas de controle. Analisaremos também como usar funções para facilitar a reutilização de software.

Resumo

- Um loop é um grupo de instruções que o computador executa repetidamente até que alguma condição de terminação seja satisfeita. Duas formas de repetição são a controlada por contador e a controlada por sentinela.
- Um contador de loops é usado para contar o número de vezes que um grupo de instruções deve ser repetido. Ele é incrementado (normalmente de 1) cada vez que o grupo de instruções é realizado.
- Geralmente, os valores sentinelas são usados para controlar a repetição quando o número exato de repetições não é conhecido de antemão e o loop inclui instruções que obtêm dados cada vez que o loop é realizado.
- Um valor sentinela é fornecido depois de todos os itens válidos de dados serem cedidos ao programa. Os valores sentinelas devem ser escolhidos com cuidado para que não haja possibilidade de serem confundidos com itens válidos de dados.
- A estrutura de repetição **for** manipula automaticamente todos os detalhes de repetições controladas por contadores. O formato geral da estrutura **for** é

for *(expressão1; expressão2; expressão3)*
instrução

- onde *expressão1* inicializa a variável de controle do loop, *expressão2* é a condição de continuação do loop e *expressão3* incrementa a variável de controle.
- A estrutura de repetição **do/while** é similar à estrutura de repetição **while,** mas a estrutura **do/while** verifica a condição de continuação do loop no final do loop, de forma que o corpo do loop é executado pelo menos uma vez. O formato da instrução **do/while** é

do
instrução **while** *(condição) ;*

- A instrução **break,** quando executada em uma das estruturas **(for, while** e **do/while),** causa a saída imediata da estrutura. A execução continua com a primeira instrução após o loop.
- A instrução **continue,** quando executada em uma das estruturas **(for, while** e **do/while),** ignora o restante das instruções no corpo da estrutura e prossegue com a próxima iteração do loop.
- A instrução **switch** manipula uma série de decisões nas quais uma determinada variável ou expressão é examinada para cada um dos valores que pode assumir, e diferentes ações são tomadas. Cada **case** em uma instrução **switch** pode fazer com que sejam executadas várias ações. Na maioria I dos programas, é necessário incluir uma instrução **break** após as instruções de cada **case,** pois do f contrário o programa executará as instruções de todos os **cases** até encontrar uma instrução **break** ou o fim da instrução **switch.** Vários **cases** podem executar as mesmas instruções reunindo os j rótulos **case** antes das instruções. A estrutura **switch** só pode examinar expressões constantes inteiras.
- A função **getchar** retorna um caractere do teclado (o dispositivo padrão de entrada) como um inteiro.

- Em sistemas UNIX e muitos outros, o caractere **EOF** é fornecido digitando a seqüência

<return> <ctrl-d>
Em VMS e DOS, o caractere **EOF** é fornecido digitando *<ctrl-z>*

- Os operadores lógicos podem ser usados para criar condições complexas combinando condições simples. Os operadores lógicos são &&, | | e !, significando E lógico, OU lógico e NÃO (negação) | lógico, respectivamente.
- Um valor verdadeiro é um valor diferente de zero.
- Um valor falso é 0 (zero).

Terminologia

break
caso default em switch char
condição de continuação do loop
condição simples
conjunto de caracteres ASCII
contador do loop
continue
corpo de um loop
<ctrl-z>
decremento
E lógico (&&)
EOF
erro off-by-one
erro por falta de um
estrutura de repetição do/while
estrutura de repetição for
estrutura de repetição while
estrutura de seleção switch
estruturas de controle aninhadas
estruturas de repetição
estruturas de única entrada/única saída
fim de arquivo
função getchar
função pow
incremento da variável de controle
justificação à direita justificação à esquerda largura do campo long
loop infinito lvalue ("left value")
negação lógica (!) operador unário
operadores lógicos OU lógico (||)
regra de aninhamento
regra de empilhamento
repetição controlada por contador
repetição definida
repetição indefinida
<return> <ctrl-d>
rótulo case rvalue ("right value")
seleção múltipla
short
sinal de subtração (menos) na justificação à esquerda tabela de verdade valor direito valor esquerdo
valor final da variável de controle
valor inicial da variável de controle
variável de controle variável de controle do loop

Erros Comuns de Programação

4.1 Como os valores em ponto flutuante podem ser valores aproximados, controlar a contagem de loops com variáveis de ponto flutuante pode resultar em valores imprecisos de contadores e exames incorretos da condição de terminação.

4.2 Usar um operador relacionai inadequado ou um valor final incorreto de um contador de loop na condição de uma estrutura while ou for pode causar erros de falta de uma repetição (off-by-one).

4.3 Usar vírgulas em vez de ponto-e-vírgula em um cabeçalho de uma estrutura for.

4.4 Colocar um ponto-e-vírgula imediatamente à direita do cabeçalho de uma estrutura for faz com que o corpo dessa estrutura seja uma instrução vazia. Normalmente isto é um erro de lógica.

4.5 Esquecer de colocar uma instrução break necessária em uma instrução switch.

4.6 Não processar caracteres de nova linha, na entrada de dados, ao ler um caractere de cada vez, pode causar erros lógicos.

4.7 Acontecem loops infinitos quando a condição de continuação do loop em uma estrutura while, for ou do/while nunca se torna falsa. Para evitar isso, certifique-se de que não há um ponto-e-vírgula imediatamente após o cabeçalho de uma estrutura while ou for. Em um loop controlado por contador, certifique-se de que a variável de controle esteja sendo incrementada (ou decrementada) no corpo do loop. Em um loop controlado por sentinela, certifique-se de que o valor sentinela seja fornecido posteriormente.

4.8 Usar o operador == para atribuição ou usar o operador = para igualdade.

Práticas Recomendáveis de Programação

4.1 Controle a contagem das repetições (loops) com valores inteiros.

4.2 *Faça um recuo nas instruções do corpo de cada estrutura de controle.*

4.3 Coloque uma linha em branco antes e depois de cada uma das principais estruturas de controle para fazer com que ela se destaque no programa.

4.4 Muitos níveis de aninhamento podem fazer com que um programa fique difícil de entender. Como regra geral, tente evitar o uso de mais de três níveis de recuos.

4.5 A combinação de espaçamento vertical antes e após as estruturas de controle e dos recuos do corpo dessas estruturas dá aos programas um aspecto bidimensional que aumenta muito a sua legibilidade.

4.6 Usar o valor final na condição de uma estrutura while ou for e usar o operador relacionai <= evitará erros por falta de uma repetição (off-by-one). Para um loop usado para imprimir os valores 1 a 10, por exemplo, a condição de continuação do loop deve ser contador <= 10 em vez de contador < 11 ou contador < 10.

4.7 Nas seções de inicialização e incremento de uma estrutura for, coloque apenas expressões que utilizem variáveis de controle. A manipulação de outras variáveis deve aparecer antes do loop (se elas devem ser executadas apenas uma vez como instruções de inicialização) ou no corpo do loop (se elas devem ser executadas uma vez para cada repetição como as instruções para incrementar ou decrementar).

4.8 Embora o valor da variável de controle possa ser modificado no corpo de um loop for, isto

pode levar a erros difíceis de perceber.

4.9 Embora instruções que sejam anteriores à estrutura for ou que façam parte do corpo da estrutura possam ser incluídas freqüentemente no cabeçalho do for, evite fazer isto porque o programa pode ficar mais difícil de ler.

4.10 Se possível, limite em uma linha o tamanho dos cabeçalhos das estruturas de controle.

4.11 Não use variáveis do tipo float e double para realizar cálculos financeiros. A imprecisão dos números de ponto flutuante pode causar erros que resultarão em valores incorretos. Nos exercícios, exploramos o uso de inteiros para realizar cálculos financeiros.

4.12 Fornecer um caso default em instruções switch. Os casos (ou cases) não-testados explicitamente em uma estrutura switch são ignorados. O caso default ajuda a evitar isto fazendo com que o programador adote um procedimento para processar condições excepcionais. Há situações nas quais não se tem necessidade do processamento default.

4.13 Embora as cláusulas case e a cláusula do caso default em uma estrutura switch possam ocorrer em qualquer ordem, é considerado uma boa prática de programação colocar a cláusula default por último.

4.14 Em uma estrutura switch em que a cláusula default está colocada por último, a instrução break não é exigida ali. Mas alguns programadores incluem este break para manter a clareza e a simetria com os outros cases.

4.15 Lembre-se de fornecer recursos de processamento para os caracteres de nova linha na entrada de dados ao processar um caractere de cada vez.

4.16 Alguns programadores sempre incluem chaves em uma estrutura do/while mesmo que elas não sejam necessárias. Isso ajuda a eliminar a ambigüidade entre a estrutura do/while contendo uma instrução e a I estrutura while.

4.17 Alguns programadores acham que break e continue violam as normas da programação estruturada. Como os efeitos dessas instruções podem ser conseguidos pelas técnicas de programação estruturada que aprenderemos em breve, esses programadores não usam break e continue.

4.18 Quando uma expressão de igualdade tem uma variável e uma constante, como em x == 1, alguns programadores preferem escrever a expressão com a constante à esquerda e o nome da variável à direita como proteção contra o erro lógico que ocorre quando o programador substitui acidentalmente o operador == pelo =.

Dicas de Performance

4.1 Em situações que possuem o desempenho como principal fator a ser considerado, onde a memória é valiosa ou a velocidade é necessária, pode ser desejável usar tamanhos menores de inteiros.

4.2 As instruções break e continue, quando usadas adequadamente, são executadas mais rapidamente que as técnicas estruturadas correspondentes que aprenderemos em breve.

4.3 Em expressões que utilizam o operador &&, coloque na extremidade esquerda a condição que tem maior probabilidade de ser falsa. Em expressões que utilizam o operador II, coloque na extremidade esquerda 1 a condição com maior probabilidade de ser verdadeira. Isso pode reduzir o tempo de execução do programa.

Dicas de Portabilidade

4.1 A combinação de teclas para entrar com **EOF** (fim do arquivo, end of file) depende do sistema.

4.2 Verificar a constante simbólica **EOF** em vez de -1 torna os programas mais portáteis. O padrão ANSI declara que **EOF** é um valor inteiro negativo (mas não necessariamente -1). Desta forma, **EOF** poderia ter diferentes valores em diferentes sistemas.

4.3 Como os ints possuem tamanhos que variam de um sistema para outro, use inteiros long se for provável o processamento de inteiros além do intervalo ±32767 e se você quiser ser capaz de executar o programa em vários sistemas computacionais diferentes.

Observação de Engenharia de Software

4.1 Há um conflito entre conseguir engenharia de software de qualidade e conseguir o software de melhor desempenho. Freqüentemente, um desses objetivos é atingido à custa do outro.

Exercícios de Revisão

4.1 Preencha as lacunas em cada uma das seguintes expressões.

a) A repetição controlada por contador também é conhecida como repetição ______________ porque sabe-se de antemão quantas vezes o loop será executado.

b) A repetição controlada por sentinela também é conhecida como repetição ______________ porque não se sabe de antemão quantas vezes o loop será executado.

c) Em uma repetição controlada por contador, utiliza-se um ______________ para contar o número de vezes que um grupo de instruções deve ser repetido.

d) A instrução ___________ , quando executada em uma estrutura de repetição, faz com que a próxima iteração do loop seja realizada imediatamente.

e) A instrução ___________ , quando executada em uma estrutura de repetição ou instrução **switch** causa a saída imediata da estrutura.

f) A ______________ é usada para examinar cada um dos valores constantes inteiros que uma determinada variável ou expressão pode assumir.

4.2 Diga se cada uma das seguintes afirmações é verdadeira ou falsa. Se a afirmação for falsa, explique por quê.

a) A estrutura de seleção **switch** exige o caso **default.**

b) O caso **default** de uma estrutura de seleção **switch** exige a instrução **break.**

c) A expressão **(x>y&&a<b)** é verdadeira se **x>y** for verdadeiro ou se **a<b** for verdadeiro.

d) Uma expressão contendo o operador I I é verdadeira se um ou ambos os operandos forem verdadeiros.

4.3 Escreva uma instrução ou um conjunto de instruções em C para realizar cada uma das seguintes tarefas:

a) Somar os inteiros ímpares entre 1 e 99 usando uma estrutura **for.** Admita que as variáveis inteiras **soma** e **contagem** já foram declarados.

b) Imprima o valor **333.546372** em um campo com **15** caracteres com precisões de **1, 2, 3, 4** e **5.** Alinhe (justifique) a saída pela esquerda. Quais os cinco valores impressos?

c) Calcule o valor de **2,5** elevado ao cubo (potência **3)** usando a função **pow.** Imprima o resultado com uma precisão de **2** em um campo com largura de **10** posições. Qual o valor impresso?

d) Imprima os inteiros de 1 a 20 usando um loop **while** e a variável **x.** Considere que a variável **x** já foi declarada, mas não inicializada. Imprima somente 5 inteiros por linha. Sugestão: Use o cálculo **x % 5.** Quando o valor desse cálculo for 0, imprima um caractere de nova linha; caso contrário, imprima um caractere de tabulação.

Repita o exercício 4.3(d) usando uma estrutura **for.**

4.4 Encontre o erro em cada um dos segmentos de código a seguir e explique como corrigi-lo.

```
a)      x = 1;
while  (x <= 10) ; x++;
}
g)      for  (y =  .1; y != 1.0; y += .1)
printf("%f\n", y);
c)      switch     (n) {
case 1:
printf("O numero e 1\n");
case 2:
printf("O numero e 2\n");
break;
```

default:
printf ("O numero nao e 1 nem 2\n"); break;
}
d) O código a seguir deve imprimir valores de 1 a 10. **n = 1;**
while (n < 10)
printf("%d ", n++);

Respostas dos Exercícios de Revisão

4.1 **a) definida. b) indefinida. c) variável de controle ou contador. d) continue. e) break. f) estrutura de seleção switch.**

4.2 a) Falso. O caso **default** é opcional. Se não forem necessárias ações default, não há necessidade de um caso **default.**
b) Falso. A instrução **break** é usada para sair da estrutura **switch.** A instrução **break** não é exigida quando o caso **default** é o último.
c) Falso. Ambas as expressões relacionais devem ser verdadeiras para que toda a expressão seja verdadeira quando o operador && é utilizado.
Verdadeiro.

4.3 a)
```
soma = 0;
for (contagem = 1; contagem <= 99; contagem += 2)
soma += contagem;
```
b)
```
printf  ("%-15.1f\n",  333.546372);  /* imprime 333.5 */
printf("%-15.2f\n",  333.546372);  /* imprime 333.55 */
printf("%-15.3f\n",  333.546372);  /* imprime 333.546 */
printf("%-15.4f\n",  333.546372);  /* imprime 333.5464 */
printf("%-15.5f\n",  333.546372);  /* imprime 333.54637 */
```
c)
```
printf("%10.2f\n", pow (2.5,  3));  /* imprime 15.63 */
```
d)
```
x = 1;
while  (x <= 20) {
printf("%d", x);
if   (x % 5  == )
printf("\n"); else
printf("\t") ; x++;
}
```
ou
```
x = 1;
while  (x <= 20)
if  (x % 5 == 0)
printf("%d\n", x++);
else
printf("%d\t", x++):
```
ou
```
x = 0;
while  (++x <= 20)
if  (x % 5 == 0)
printf("%d\n",  x); else
printf("%d\t", x)
```
49.
```
for (x = 1; x <= 20; x++) {
printf("%d",  x); if  (x % 5 == 0)
printf("\n"); else
}        printf("\t");
```
ou
```
for (x = 1; x <= 20; x++) if  (x % 5 == 0)
printf("%d\n", x); else
printf("%d\t", x);
```

4.4 a) Erro: O ponto-e-vírgula depois do cabeçalho do **while** causa um loop infinito.
Correção: Substituir o ponto-e-vírgula por um { ou remover tanto o ; como o }

b) Erro: Usar um número de ponto flutuante para controlar uma estrutura de repetição **for** Correção: Usar um número inteiro e realizar o cálculo adequado para obter os valores desejados.

```
for  (y = 1;  y  != 10; y++)
printf("%f\n",(float)y /10);
```

c) Erro: Falta a instrução **break** nas instruções do primeiro **case.**
Correção: Adicionar uma instrução **break** no final das instruções do primeiro **case.** Note que isto não é necessariamente um erro se o programador quiser que a instrução do **case 2** : seja executada sempre que a instrução de **case 1**: for executada.

2. Erro: Operador relacionai inadequado na condição de repetição do **while.**
   Correção: Usar <= em vez de <.

Exercícios

4.4 Ache o erro em cada uma das opções a seguir: (Obs.: pode haver mais de um erro.)

a) For (x = 100, x >= 1, x++)
printf("%d\n" , x);
b) O código a seguir deve imprimir se o número dado é ímpar ou par:
switch (valor *% 2) {*
case 0:
printf("Inteiro par\n");
case 1:
printf ("Inteiro impar\n"); }
c) O código a seguir deve receber como entrada de dados um inteiro e um caractere e imprimi-los. Admita que o usuário digitou **100 A** como entrada de dados.
scanf("%d", &intVal); charVal = getchar ( );
printf("Inteiro: %d\nCaractere: %c\n", intVal, charVal);
d) for (x = .000001; x <= .0001; x += .000001)
printf("%.7f\n", x);
e) O código a seguir deve fornecer como saída os inteiros ímpares de 999 até 1: **for (x = 999; x >= 1; x += 2)**
printf("%d\n", x);
f) O código a seguir deve fornecer como saída os inteiros pares de 2 até 100: **contador = 2;**
Do {
if (contador *% 2 ==* **0)**
printf ("%d\n", contador);
contador += 2; } While (contador < 100);

4.5 O código a seguir deve somar os inteiros de 100 até 150 (admita que **total** foi inicializado como 0):
for (x = 100; x <= 150; x+ +);
total += x;

4.6 Diga que valores da variável de controle **x** são impressos por cada uma das seguintes instruções:

a) for(x = 2; x <= 13; x+= 2)
printf("%d\n", x);
b) for(x = 5; x <= 22; x+= 7)
printf(*"%â\n"***, x);**
c) for(x = 3; x <= 15; x+= 3)
printf("%d\n", x);
d) for(x = 1; x <= 5; x+= 7)
printf("%d\n", x);
e) for(x = 12; x <= 2; x -= 3)
printf("%d\n", x);

4.7 Escreva instruções **for** que imprimam as seguintes series de valores:

a) 1,2,3,4,5,6,7
b) 3,8,13,18,23
c) 10,14,8,2,-4,-10
d) 19, 27,35,43,51

4.8 O que o seguinte programa faz?

```
#include <stdio.h>
main ) {
int i,  j, x, y;
printf("Entre com inteiros no intervalo 1-20: "); scanf("%d%d", &x, &y) ;
for  (i = 1;  i <= y;   i++) {
for  (j  = 1;   j <= x; j++) printf("@") ;
printf("\n");
}
return 0;

}
```

4.9 Escreva um programa que some uma seqüência de inteiros. Admita que o primeiro inteiro lido com scanf especifica o número de valores que ainda devem ser fornecidos. Seu programa deve ler apenas um valor cada vez que scanf for executado. Uma seqüência típica de entrada poderia ser
5 100 200 300 400 500
onde o 5 indica que os valores subseqüentes 5 devem ser somados.

4.10 Escreva um programa que calcule e imprima a média de vários inteiros. Admita que o último valor lido com scanf é o sentinela 9999. Uma seqüência típica de entrada poderia ser
10 8 11 7 9 9999
indicando que a média de todos os valores que aparecem antes de 9999 deve ser calculada.

4.11 Escreva um programa que encontre o menor valor entre vários inteiros. Admita que o primeiro valor lido especifica o número de valores restantes.

4.12 Escreva um programa que calcule e imprima a soma dos inteiros pares de 2 a 30.

4.13 Escreva um programa que calcule e imprima o produto dos inteiros ímpares de 1 a 15.

4.14 A *função fatorial é* usada com freqüência em problemas de probabilidade. O fatorial de um número positivo *n* (escrito *n!* e pronunciado *"fatorial de n") é* igual ao produto dos inteiros positivos de 1 a *n*. Escreva um programa que calcule os fatoriais dos inteiros de 1 a 5. Imprima o resultado no formato de uma tabela. Que problema pode evitar que você possa calcular o fatorial de 20?

4.15 Modifique o programa de juros compostos da Seção 4.6 para repetir seus passos para taxas de juros de 5 por cento, 6 por cento, 7 por cento, 8 por cento, 9 por cento e 10 por cento. Use um loop for para variar a taxa de juros.

4.16 Escreva um programa que imprima os seguintes padrões separadamente, um abaixo do outro. Use loops for para gerar os padrões. Todos os asteriscos (*) devem ser impressos por uma única instrução printf na forma printf ("*"); (isto faz com que os asteriscos sejam impressos lado a lado). Sugestão: Os dois últimos padrões exigem que cada linha comece com um número adequado de espaços em branco.
(A) (B) (C) (D)

4.17 Economizar dinheiro se torna muito difícil em períodos de recessão, portanto as empresas podem reduzir seus limites de crédito para evitar que suas contas a receber (dinheiro que lhes é devido) se tornem muito grandes. Em resposta a uma recessão prolongada, uma companhia reduziu o limite de crédito de seus clientes à metade. Desta forma, se um cliente tinha um limite de crédito de $2000, agora seu limite passou a ser | $1000. Se um cliente tinha um limite de crédito de $5000, seu limite passou a ser $2500. Escreva um programa que analise a situação do crédito de três clientes dessa companhia. De cada cliente você recebe
1. O número de sua conta
2. Seu limite de crédito antes da recessão
3. Seu saldo atual (i.e., a quantia que o cliente deve à companhia).

Seu programa deve calcular e imprimir o novo limite de crédito de cada cliente e determinar (e imprimir) | que clientes possuem saldos que excedem seus novos limites de crédito.
4.10

4.18 Uma aplicação interessante dos computadores é para desenhar gráficos de linhas e de barras (chamados algumas vezes "histogramas"). Escreva um programa que leia cinco números (cada um deles entre 1 e 30). Para cada número lido, seu programa deve imprimir uma linha contendo aquele número de asteriscos adjacentes. Por exemplo, se seu programa ler o número sete, deve imprimir *******.

4.19 Uma loja de venda de produtos por reembolso postal vende cinco produtos diferentes cujos preços de varejo são mostrados na tabela seguir:

Número do produto	**Preço de varejo**
1	**$ 2,98**
2	**4,50**
3	**9,98**
4	**4,49**
5	**6,87**

Escreva um programa que leia uma série de pares de números como se segue:
10 Número do produto
11 Quantidade vendida em um dia

Seu programa deve usar uma instrução **switch** para ajudar a determinar o preço de varejo de cada produto. Seu programa deve calcular e mostrar o valor total a varejo de todos os produtos vendidos na semana passada.

4.20 Complete as seguintes tabelas de verdade preenchendo cada espaço em branco com 0 ou 1.

expressão1	**expressão2**	**expressão 1 && expressão2**
0	**0**	**0**
0	**Diferente de zero**	**0**
Diferente de zero	**0**	**_**
Diferente de zero	**Diferente de zero**	**_**

expressão1	**expressão2**	**expressão 1 | | expressão2**
0	**0**	**0**
0	**Diferente de zero**	**0**
Diferente de zero	**0**	**_**
Diferente de zero	**Diferente de zero**	**_**

expressão	**!expressão**
0	**1**
Diferente de zero	**_**

4.21 Escreva novamente o programa da Fig. 4.2 de modo que a inicialização da variável **contador** seja feita na declaração e não na estrutura **for.**

4.22 Modifique o programa da Fig. 4.7 de forma que ele calcule o grau médio da turma.

4.23 Modifique o programa da Fig. 4.6 de forma que ele use somente inteiros para calcular os juros compostos. (Sugestão: Trate todas as quantias monetárias como números inteiros de pennies, sabendo que um penny é um centésimo de um dólar. A seguir, "divida" o resultado em suas partes de dólares e cents usando as operações de divisão e resto, respectivamente. Insira um ponto.)

4.24 Admita que **i =** 1, **j =** 2, **k** = 3 e m = 2. O que cada uma das seguintes instruções imprime?

a) printf("%d", i == 1);
b) printf("%d", j == 3);
c) printf("%d", i >= 1 && j < 4);
d) printf("%d", m <= 99 && k < m);
e) printf("%d", j >= i I I k == m);
f) printf***("%d",*** k + m < j || 3 - j >= k) ;
g) printf("%d", !m);
h) printf("%d", ! (j - m));
i) printf ("%d", ! (k > m)) ;
j) printf("%d", ! (j > k));

4.25 Imprima uma tabela de valores equivalentes decimais, binários, octais e hexadecimais. Se você não está familiarizado com estes sistemas de numeração, leia inicialmente o Apêndice D se quiser tentar solucionar este exercício.

4.26 Calcule o valor de ir a partir da série infinita
. 4,4 4,4 4 , 3 5 7 9 11
Imprima uma tabela que mostra o valor de (pi) aproximado por 1 termo dessa série, por dois termos, por três termos etc. Quantos termos dessa série são necessários para obter 3.14? 3.141? 3.1415? 3.14159?

4.27 *(Números de Pitágoras)* Um triângulo retângulo pode ter lados que sejam todos inteiros. O conjunto de três valores inteiros para os lados de um triângulo retângulo é chamado números de Pitágoras. Esses três lados j devem satisfazer o relacionamento de que a soma dos quadrados dos dois lados (catetos) deve ser igual ao quadrado da hipotenusa. Encontre todos os números de Pitágoras para cateto1, cateto2 e hipotenusa menores que 500. Use três loops for aninhados que simplesmente experimentem todas as possibilidades. Este é um exemplo de cálculo por "força bruta". Ela não é esteticamente agradável para muitas pessoas. Mas há muitas razões para estas técnicas serem importantes. Em primeiro lugar, com o poder da computação crescendo a um ritmo tão acelerado, as soluções que levariam anos ou mesmo séculos de tempo de cálculo para serem produzidas com a tecnologia de apenas alguns anos atrás podem ser produzidas atualmente em horas, minutos ou mesmo segundos. Os chips recentes de microprocessadores podem processar mais de 100 milhões de instruções por segundo! E é provável que chips capazes de processar bilhões de instruções por segundo apareçam ainda nos anos 90. Em segundo lugar, como você aprenderá em cursos mais avançados de ciência da computação, há um número grande de problemas interessantes para os quais não há método conhecido para um algoritmo diferente daquele utilizado pela força bruta. Investigamos muitos tipos de metodologias para resolução de problemas neste livro. Levaremos em consideração muitos métodos de força bruta em vários problemas interessantes.

4.28 Uma empresa paga seus empregados como gerentes (que recebem um salário fixo mensal), trabalhadores comuns (que recebem um salário fixo por hora para as primeiras 40 horas de trabalho e 1,5 vez seu salário por hora normal para as horas extras trabalhadas), trabalhadores por comissão (que recebem $250 mais 5,7% de suas vendas brutas) ou trabalhadores por empreitada (que recebem uma quantia fixa por item para cada um dos itens produzidos — cada trabalhador por empreitada dessa empresa trabalha com apenas um tipo dei item). Escreva um programa que calcule o pagamento semanal de cada empregado. Você não sabe de antemão o número de empregados. Cada tipo de empregado tem seu código próprio de pagamento: gerentes possuem o código 1, trabalhadores comuns, o código 2, trabalhadores por comissão, o código 3, e trabalhadores por empreitada, o código 4. Use switch para calcular o pagamento de cada empregado com base em seu código de pagamento. Dentro do switch peça ao usuário (i.e., o responsável pela folha de pagamento) para entrar com os fatos adequados à necessidade de seu programa para calcular o pagamento de cada empregado com base em seu código.

4.29 *(Leis de De Morgan)* Neste capítulo, analisamos os operadores lógicos &&, II e !. As Leis de De Morgan podem fazer com que algumas vezes nos seja mais conveniente exprimir uma expressão lógica. Estas lei afirmam que a expressão ! (*condição 1 && condição2) é* logicamente equivalente à expressão *(!condição1* I I ! *condição2).* Além disso, a expressão ! (*condição1* || *condição2)* é logicamente equilente à expressão (! *condição1* || ! *condição2).* Use as Leis de De Morgan para escrever expressões equivalentes a cada um dos itens seguintes e depois escreva um programa que mostre que a expressão original e a nova expressão em cada caso são equivalentes.

a) ! (x < 5) && ! (y >= 7)
b) !(a == b) II !(g != 5)
c) !((x <= 8) && (Y > 4))
d) !((i > 4) II (j <= 6))

4.30 Escreva novamente o programa da Fig. 4.7 substituindo a instrução switch por uma instrução if/**else** aninhada: tenha o cuidado de lidar adequadamente com o caso default. Depois escreva novamente esta nova versão substituindo a instrução if/else aninhada por uma série de instruções if; aqui, também, tenha o cuidado de lidar apropriadamente com o caso default (isso é mais difícil do que com a versão if/else aninhada). Este exercício demonstra que switch é uma comodidade e que qualquer instrução switch pode ser escrita apenas com instruções de seleção simples. 431

4.31 Modifique o programa escrito no Exercício 4.31 para ler um número ímpar de 1 a 19 para especificar o número de linhas no losango. Seu programa deve então exibir um losango do tamanho apropriado.

4.32 Se você está familiarizado com numerais romanos, escreva um programa que imprima uma tabela de todos os números romanos equivalentes aos números decimais de 1 a 100.

4.33 Escreva um programa que imprima uma tabela de números binários, octais e hexadecimais equivalentes aos números decimais no intervalo de 1 a 256. Se você não estiver familiarizado com esses sistemas de numeração, leia primeiro o Apêndice D se é seu desejo tentar fazer este exercício.

4.34 Descreva o processo que você gostaria de usar para substituir um loop do/while por um loop while equivalente. Que problema ocorre quando se tenta substituir um loop while por um loop do/while equivalente? Suponha que você deve remover um loop while e substituí-lo por um do/while. Que estrutura adicional de controle seria necessária utilizar e como ela seria usada para assegurar que o programa resultante se comporte exatamente como o original?

4.35 Escreva um programa que receba como entrada o ano no intervalo de 1994 a 1999 e use o loop for para produzir um calendário condensado e impresso de uma forma organizada. Tome cuidado com os anos bissextos.

4.36 Uma crítica às instruções break e continue é que cada uma delas não é estruturada. Realmente as instruções break e continue sempre podem ser substituídas por instruções estruturadas, embora fazer isto possa ser esquisito. Descreva de uma forma geral como qualquer instrução break poderia ser removida de um loop de um programa e substitua essa instrução por alguma instrução estruturada equivalente. (Sugestão: A instrução break abandona um loop de dentro de seu corpo. A outra maneira de fazer isto é impondo uma falha no teste de continuação do loop. Pense em utilizar, no teste de continuação do loop, um segundo teste que indique "saída prematura devido a uma condição idêntica ao break".) Use a técnica desenvolvida aqui para remover a instrução break do programa da Fig. 4.11.

4.37 Que faz o seguinte segmento de programa?

```
for (i = 1;  i <= 5;  i++) {
for  (j = 1; j <= 3; {
for  (k = 1,  k <= 4; k++)
printf("*") ;
printf("\n");
}
```

printf("\n"   );

4.38 Descreva de uma maneira geral como você removeria todas as instruções continue de um loop de um programa e substituiria essas instruções por outras instruções estruturadas equivalentes. Use a técnica desenvolvida aqui para remover a instrução continue do programa da Fig. 4.12.

4.39 Descreva de uma maneira geral como você removeria as instruções break de uma estrutura switch e as substituiria por instruções estruturadas equivalentes. Use a técnica (talvez estranha) desenvolvida aqui para remover as instruções break do programa da Fig. 4.7.

4.40 Escreva um programa que imprima o seguinte losango. Você pode usar instruções printf que imprimam tanto um único asterisco como um único espaço em branco. Maximize seu uso de repetições (com estruturas for aninhadas) e minimize o número de instruções printf.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5

Funções

Objetivos

- Entender como construir programas em módulos a partir de pequenas partes chamadas funções.
- Apresentar as funções matemáticas comuns disponíveis na biblioteca padrão do C
- Poder criar novas funções.
- Entender os mecanismos usados para passar informações entre funções.
- Apresentar técnicas de simulação usando geração aleatória de números.
- Entender como escrever e usar funções que chamem a si mesmas.

Uma forma sempre vem após uma função.
Louis Henri Sullivan

Epluribus unum. (Unidade constituída de muitos.)
Virgílio

Ohl Lembre-se do passado, Proclame a volta do tempo.
William Shakespeare
Ricardo III

Chame-me Ismahel.
Herman Melville
Moby Dick

Quando você me chamar disso, sorria.
Owen Wister

Sumário

5.1 Introdução

A maioria dos programas de computador que resolvem problemas do mundo real é muito maior do que os programas apresentados nestes primeiros capítulos. A experiência tem mostrado que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas partes ou *módulos*, sendo cada uma delas mais fácil de manipular do que o programa original. Essa técnica é chamada *dividir e conquistar*. Este capítulo descreve os recursos da linguagem C que facilitam o projeto, implementação, operação e manutenção de programas grandes.

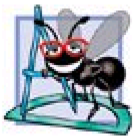
5.2 Módulos de Programas em C

Os Módulos em C são chamados/funções. Geralmente, os programas em C são escritos combinando novas funções que o programador escreve com funções "prontas" disponíveis na *biblioteca padrão do C (C* standart *library).* Analisaremos ambos os tipos de funções neste capítulo. A biblioteca padrão do C fornece um excelente conjunto de funções para realizar cálculos matemáticos comuns, manipulação de strings, manipulação de caracteres, entrada/saída e muitas outras operações. Isso torna mais fácil o trabalho do programador porque essas funções fornecem muitos recursos de que ele precisa.



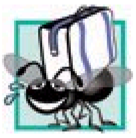
Boa prática de programação 5.1

Familiarize-se com o ótimo conjunto de funções da biblioteca padrão do ANSI C.



Observação de engenharia de software 5.1

Evite reinventar a roda. Quando possível, use as funções da biblioteca padrão do ANSI C em vez de escrever novas funções. Isso reduz o tempo de desenvolvimento de programas.



Dicas de portabilidade 5.1

Usar as funções da biblioteca padrão do ANSIC ajuda a tornar os programas mais portáteis.

Embora as funções da biblioteca padrão não sejam tecnicamente parte da linguagem C, são sempre fornecidas com os sistemas ANSI C. As funções **printf, scanf** e **pow** que usamos nos capítulos anteriores são funções da biblioteca padrão.

O programador pode escrever funções para definir tarefas específicas e que podem ser utilizadas em muitos locais dos programas. Algumas vezes elas são chamadas *funções definidas pelo programador.* As instruções reais que definem a função são escritas apenas uma vez e são escondidas das outras funções.

As funções são *ativadas (chamadas* ou *invocadas)* por uma *chamada de função.* A chamada da função especifica o nome da função e fornece informações (como *argumentos)* de que a referida função necessita para realizar a tarefa designada. Uma analogia comum para isso é a forma hierárquica de administração.

O chefe (a *função que chama* ou *chamadora)* pede a um funcionário (ou subordinado, ou seja, a *função que é chamada)* que realize uma tarefa e informe quando ela for concluída. Por exemplo, uma função que deseje exibir informações na tela chama a função **printf** para realizar aquela tarefa, a seguir, **printf** exibe as informações e avisa — ou *retorna* — à função que fez a chamada quando a tarefa for concluída. A função chefe não sabe como a função subordinada realizou suas tarefas. O subordinado pode chamar outras funções e o chefe não estará ciente disso.

Em breve veremos como essa "ocultação" dos detalhes de implementação favorece a boa engenharia de software. A Fig. 5**.1** mostra a função **main (principal)** se comunicando com várias funções subordinadas de forma hierárquica. Observe que **subordinada1** age como uma função chefe para **subordinada4** e **subordinada5.** Os relacionamentos entre funções podem ter estruturas diferentes da mostrada nessa figura.

5.3 Funções da Biblioteca Matemática

As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns. Usamos várias funções matemáticas aqui para introduzir o conceito de funções. Posteriormente no livro, analisaremos muitas das outras funções da biblioteca padrão do C. Uma lista completa das funções da biblioteca padrão do C é fornecida no Apêndice A.

As funções são usadas normalmente em um programa escrevendo o nome da função seguido pelo parêntese esquerdo, pelo *argumento* (ou uma lista de argumentos separada por vírgulas) da função e pelo parêntese direito. Por exemplo, um programador que desejasse calcular e imprimir a raiz quadrada de **900.0** poderia escrever

```
printf("%.2f",  sqrt(900.0));
```

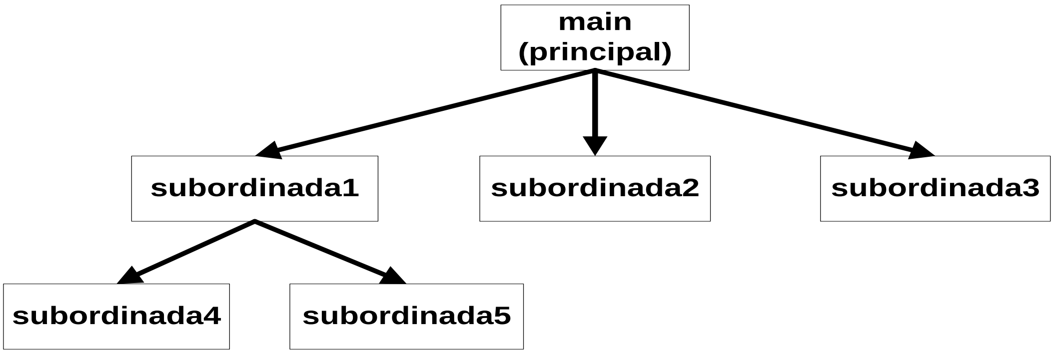


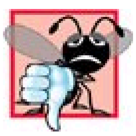
Fig. 5.1 Relacionamento hierárquico função chefe/função subordinada.

Quando essa instrução é executada, a função **sqrt** da biblioteca matemática é chamada para calcular a raiz quadrada do número contido entre os parênteses **(900.0).** O número **900.0** é o argumento da função **sqrt.** A instrução anterior imprimiria **30.00**. A função **sqrt** utiliza um argumento do tipo **double** e retorna um resultado do tipo **double.** Todas as funções da biblioteca matemática retornam o tipo de dado **double.**



Boa prática de programação 5.2

Inclua o arquivo de cabeçalho (header) matemático usando a diretiva do pré-processador #include <math.h> ao usar funções da biblioteca matemática.



Erro comun de programação 5.1

Esquecer de incluir o arquivo de cabeçalho matemático ao usar funções da biblioteca matemática pode levar a resultados errados.

Os argumentos de funções podem ser constantes, variáveis, ou expressões. **Sec1 = 13.0 , d = 3.0 e f = 4.0,** a instrução

```
printf("%.2f",  sqrt(c1 + d * f));
```

calcula e imprime a raiz quadrada de **13.0 + 3.0*4.0 = 25.0,** que é **5.00.**

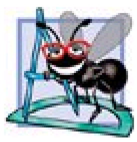
Algumas funções da biblioteca matemática do C estão resumidas na Fig. 5.2. Na figura, as variáveis **x** e **y** são do tipo **double.**

5.4 Funções

Função	Descrição	Exemplo
sqrt(x)	Raiz quadrada de x	sqrt(900.0) é 30 sqrt(9.0) é 10
exp(x)	Função exponecial de e^x	exp(1.0) é 2.718282 exp(2.0) é 7.389056
log(x)	Logaritmo natural de x (base e)	log (2.718282) é 1.0 log (7.389056) é 2.0
log10(x)	Log de x (base 10)	log10(1.0) é 0.0 log10(10.0) é 1.0 log10(100.0) é 10.0
fabs(x)	Valor absoluto de x	Se x > 0 então fabs(x) é x Se x = 0 então fabs(x) é 0.0 Se x < 0 então fabs(x) é –x
ceil(x)	Arredonda x para o menor inteiro maior que x	ceil(9.2) é 10 ceil(-9.8) é -9
floor (x)	Arredonda x para o maior inteiro menor que x	floor(9.2) é 9 floor(-9.8) é -10
pow(x,y)	x elevado a potencia y	pow(2,7) é 128.0 pow(9, .5) é 3.0
fmod(x,y)	Resto de x/y, como numero de ponto flutuante	fmod(13.657, 2.333_ é 1.992
sin(x)	Seno trigonométrico de x em rad	sin(0.0) é 0
cos(x)	Cosseno trigonométrico de x em rad	tos(0.0) é 1
tan(x)	Tangente trigonométrico de x em rad	tan(0.0) é 0

Fig. 5.2 Funções da biblioteca matemática usadas normalmente.

As funções permitem ao programador modularizar um programa. Todas as variáveis declaradas em definições de funções são *variáveis locais* — elas são conhecidas apenas na função onde são definidas. A maioria das funções tem uma lista de *parâmetros.* Os parâmetros fornecem os meios para a comuni-ío de informações entre funções. Os parâmetros de uma função também são variáveis locais.



Observação de engenharia de software

Em programas contendo muitas funções, ***main (principal)*** *deve ser implementada como um grupo de chamadas a funções que realizam o núcleo do trabalho do programa.*

Há vários motivos para "funcionalizar" um programa. O método dividir-para-conquistar torna o de-senvolvimento do programa mais flexível. Outra motivação é a *capacidade de reutilização (reusability) do software* — usar funções existentes como blocos pré-fabricados para criar novos programas. A capa-cidade de reutilização do software é um fator importante para a programação orientada a objetos. Com j boa definição e nomenclatura das funções, os programas podem ser criados a partir de funções padroni-zadas que realizem tarefas específicas, em vez de serem construídos usando código padronizado. Essa técnica é conhecida como *abstração.* Usamos a abstração sempre que escrevemos programas que inclu-em funções como printf, **scanf e pow.** Um terceiro motivo é para evitar repetição de código em um programa. Incluir código em uma

função permite que ele seja executado em vários locais de um programa simplesmente chamando a função.

Observação de engenharia de software 5.3

Cada função deve se limitar a realizar uma tarefa simples e bem-definida, e o nome da função deve expressar efetivamente aquela tarefa. Isso facilita a abstração e favorece a capacidade de reutilização do software.

Observação de engenharia de software 5.4

Se você não puder escolher um nome conciso que expresse o que a função faz, é possível que sua função esteja tentando realizar muitas tarefas diferentes. Normalmente é melhor dividir tal função em várias funções menores.

5.5 Definições de Funções

Cada programa que apresentamos consistiu em uma função denominada **main** que chamou as funções j da biblioteca padrão para realizar suas tarefas. Agora vamos analisar como os programadores escrevem j suas próprias funções personalizadas.

Considere um programa que use a função **square** para calcular os quadrados dos números inteiros de 1 a 10 (Fig. 5.3).

```
/* Funcao square definida pelo programador */
#include <stdio.h>
int square(int);        /* protótipo da funcao */
main() {
int x;
for  (x = 1;  x <= 10; x++)
printf("%d   ",  square(x));

return 0;
}
/* Definição da funcao */
int square(int y)
{
return y * y;
}
```

```
1   4   9   16   25   36   49   64   81   100
```

Fig. 5.3 Usando uma função definida pelo programador,



Boa prática de programação 5.3

Coloque uma linha em branco entre as definições das funções para separá-las e melhorar a legibilidade do programa.

A função **square** é *chamada* ou *invocada* em **main** dentro da instrução **printf:**

printf("%d ", square(x));

A função **square** recebe uma cópia do valor de **x** no *parâmetro* **y.** Então **square** calcula **y * y.** O resultado é passado de volta para a função **printf** em **main** onde **square** foi chamada e **printf** exibe o resultado. Esse processo é repetido dez vezes usando a estrutura de repetição **for.**

A definição de **square** mostra que essa função espera receber um parâmetro inteiro **y.** A palavra-chave **int** antes do nome da função indica que **square** retorna um resultado inteiro. A instrução **return** em **square** passa o resultado do cálculo de volta para a função chamadora.

A linha

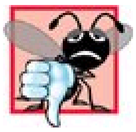
int square(int);

é um protótipo da função. O **int** entre parênteses informa ao compilador que **square** espera receber um valor inteiro da função que faz a chamada. O **int** à esquerda do nome da função **square** informa ao compilador que **square** retorna um resultado inteiro à função que faz a chamada. O compilador consulta o protótipo da função para verificar se as chamadas de **square** contêm o tipo correto do valor de retorno, o número correto de argumentos, os tipos corretos dos argumentos e se os argumentos estão **na** ordem correta. Os protótipos de funções são analisados com mais detalhes na Seção 5.6.

O formato de uma definição de função é

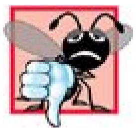
tipo-do-valor-de-retorno nome-da-função (lista-de-parâmetros) **{**
declarações instruções
}

O nome-da-função é qualquer identificador válido. O *tipo-do-valor-de-retorno é* o tipo dos dados do resultado devolvido à função que realiza a chamada. O *tipo-do-valor-de-retorno* **void** indica que a função não devolve um valor. Um *tipo-de-valor-de-retorno* não-especificado é sempre assumido pelo compila-dor como sendo **int.**



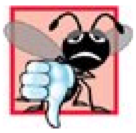
Erro comun de programação 5.2

*Omitir o tipo-do-valor-de-retorno em uma definição de função causa um erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int.***



Erro comun de programação 5.3

Esquecer de retornar um valor de uma função que deve fazer isso pode levar a erros inesperados. O padrão ANSI estabelece que o resultado dessa omissão não é definido.



Erro comun de programação 5.4

*Retornar um valor de uma função cujo tipo de retorno foi declarado **void** causa um erro de sintaxe.*

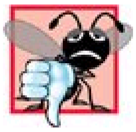


Boa prática de programação 5.4

*Embora a omissão de um tipo de retorno seja assumido como **int,** sempre declare explicitamente o tipo de retorno. Entretanto, o tipo de retorno para **main** é normalmente omitido*

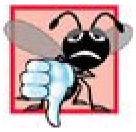
A *lista-de-parâmetros* é uma lista separada por vírgulas contendo as declarações dos parâmetros re-cebidos pela função quando ela é chamada. Se uma função não recebe nenhum valor, a *lista-de-parâme-**tros*** é **void.** Deve ser listado explicitamente um tipo para

cada parâmetro, a menos que o parâmetro peja do tipo **int.** Se um tipo não for listado, **int** é assumido.



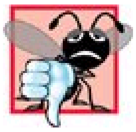
Erro comun de programação 5.5

*Declarar parâmetros da função do mesmo tipo como **float x, y** em vez de **float x, float y.** A declaração de parâmetros **float x, y** tornaria na realidade **y** um parâmetro do tipo **int** porque **int** é o default.*



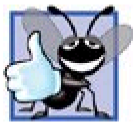
Erro comun de programação 5.6

Colocar um ponto-e-vírgula após o parêntese direito ao encerrara lista de parâmetros de uma definição de função é um erro de sintaxe.



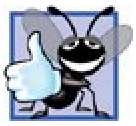
Erro comun de programação 5.7

Definir um parâmetro de função novamente como variável local dentro da função é um erro de sintaxe.



Boa prática de programação 5.5

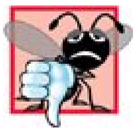
*Inclua o tipo de cada parâmetro na lista de parâmetros, mesmo que aquele parâmetro seja do tipo default **int.***



Boa prática de programação 5.6

Embora não seja incorreto fazer isso, não use os mesmos nomes para os argumentos passados para uma função e os parâmetros correspondentes na definição da função. Isso ajuda a evitar ambigüidade.

As *declarações* e as *instruções* entre chaves formam o *corpo da função,* também chamado *bloco.* Um bloco é simplesmente uma instrução composta que inclui declarações. As variáveis podem ser declaradas em qualquer bloco e os blocos podem estar aninhados. *Uma função não pode ser definida no interior de outra função sob quaisquer circunstâncias.*



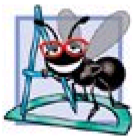
Erro comun de programação 5.8

Definir uma função dentro de outra função é um erro de sintaxe.



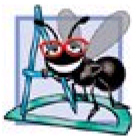
Boa prática de programação 5.7

Escolher nomes de função e parâmetros significaivos torna os programas mais legíveis e ajuda a evitar o uso excessivo de comentários.



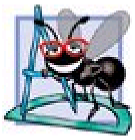
Observação de engenharia de software 5.5

Uma função não deve ser maior do que uma página. Melhor ainda, uma função não deve ser maior do que metade de uma página. Funções pequenas favorecem a capacidade de reutilização do software.



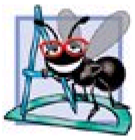
Observação de engenharia de software 5.6

Os programas devem ser escritos como conjuntos de pequenas funções. Isso torna os programas mais fáceis de escrever, depurar, manter e modificar.



Observação de engenharia de software 5.7

Uma função que exige um grande número de parâmetros pode estar realizando tarefas demais. Pense na possibilidade de dividir a função em funções menores que realizem tarefas separadas. O cabeçalho da função deve estar contido em uma linha, se possível.



Observação de engenharia de software 5.8

O protótipo da função, o arquivo de cabeçalho e as chamadas da função devem concordar quanto ao número, tipo e ordem dos argumentos e parâmetros e também quanto ao tipo do valor de retorno.

Há três maneiras de retornar controle ao ponto no qual uma função foi chamada. Se a função não fornecer um valor como resultado, o controle é retornado simplesmente quando a chave que indica o término da função é alcançada, ou executando a instrução

return;

Se a função fornecer um valor como resultado, a instrução

return *expressão;*

retorna o valor de *expressão* à função que realizou a chamada.

Nosso segundo exemplo usa uma função definida pelo programador denominada **maximum** para determinar e retornar o maior entre três inteiros (Fig. 5.4). Os três inteiros são fornecidos pelo usuário por meio da instrução **scanf**. A seguir, os inteiros são passados para **maximum** que determina o maior inteiro. Esse valor é retornado ao programa principal (main) por meio da instrução **return** em **maximum.** ***O*** valor retornado é atribuído à variável **maior** que é então impressa na instrução **printf.**

```c
/* Encontrar o maior de tres inteiros */
#include <stdio.h>
int maximum(int,  int,  int);  /* protótipo da funcao */
main() {
int a, b, c;
printf("Entre com tres inteiros: ");
scanf("%d%d%d", &a, &b, &c);
printf("0 maior e: %d\n", maximum(a, b, c));
return 0;
}
/* Definição da funcao maximum */
int maximum(int x,  int y,  int z)
{
int max = x;
if  (y > max)
    max = y;
if  (z > max)
    max = z;
return max;
}
```

Entre com tres inteiros: 22 85 17 O maior e: 85

Entre com tres inteiros: 85 22 17 O maior e: 85

Entre com tres inteiros: 22 17 85 O maior e: 85

Fig. 5.4 Função **maximum** definida pelo programador.

5.6 Protótipos de Funções

Um dos recursos mais importantes do ANSI C é o *protótipo de função.* Esse recurso foi adquirido dos desenvolvedores do C++ pelo comitê do ANSI C. Um protótipo de função diz ao compilador o tipo do dado retornado pela função, o número de parâmetros que a função espera receber, os tipos dos parâme-tros e a ordem na qual esses parâmetros são esperados. O compilador usa protótipos de funções para validar as chamadas de funções. As versões anteriores do C não realizavam esse tipo de verificação, portanto era possível chamar funções impropriamente sem que o compilador detectasse os erros. Tais chamadas poderiam levar a erros fatais de execução ou a erros não-fatais que causavam erros lógicos sutis e difíceis de detectar. Os protótipos de funções do ANSI C corrigem essa deficiência.



Boa prática de programação 5.8

Inclua protótipos de todas as funções, afim de tirar proveito dos recursos de verificação do C. Use as diretivas de pré-processador ***#include*** *para obter protótipos de todas as funções da biblioteca padrão a partir dos arquivos de cabeçalho das bibliotecas apropriadas. Use também* ***#include*** *para obter os arquivos de ca-beçalho que contêm os protótipos de funções usados por você ou pelos membros de sua equipe.*

O protótipo da função **maximum** na Fig. 5.4 é

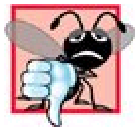
int maximum(int, int, int);

Esse protótipo de função declara que **maximum** utiliza três argumentos do tipo **int** e retorna um resultado do tipo **int.** Observe que o protótipo da função é idêntico à primeira linha da definição da função **maximum** exceto que os nomes dos parâmetros **(x, y** e **z)** não são incluídos.



Boa prática de programação 5.9

Algumas vezes, os nomes dos parâmetros são incluídos nos protótipos de funções para fins de documentação. O compilador ignora esses nomes.



Erro comun de programação 5.9

Esquecer o ponto-e-vírgula no final de um protótipo de função causa um erro de sintaxe.

Uma chamada de função que não corresponda a um protótipo de função causa um erro de sintaxe. Também é gerado um erro se o protótipo da função e sua definição discordarem. Por exemplo, na Fig. 5.4, se o protótipo da função tivesse sido escrito como

```
void maximum(int,  int,  int);
```

o compilador geraria um erro porque o tipo de retorno **void** no protótipo da função diferiria do tipo de retorno **int** no cabeçalho da função.

Outro recurso importante dos protótipos de funções é a *coerção de argumentos,* i.e., a imposição de argumentos do tipo apropriado. Por exemplo, a função **sqrt** da biblioteca matemática pode ser chamada com um argumento inteiro, muito embora o protótipo da função em **math.h** especifique um argumento **double,** e funcionará corretamente. A instrução

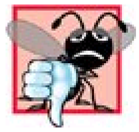
```
printf("%.3f\n", sqrt(4));
```

calcula corretamente **sqrt (4)** e imprime o valor **2.000**. O protótipo da função faz com que o compilador converta o valor inteiro **4** para o valor **double 4.0** antes de o mesmo ser passado para **sqrt.** Em geral, os valores de argumentos que não correspondem precisamente aos tipos de parâmetros no protótipo da função são convertidos para o tipo apropriado antes de a função ser chamada. Essas con-versões podem levar a resultados incorretos se as *regras de promoção* do C não forem cumpridas. As| regras de promoção especificam como alguns tipos podem ser convertidos para outros sem perda dos dados. Em nosso exemplo **sqrt** anterior, um **int** é convertido automaticamente em **double** sem modificação de seu valor. Entretanto, um **double** convertido a um **int** tranca a parte fracionária do valor **double.** Converter tipos inteiros grandes para tipos inteiros pequenos (e.g., **long** para **short)** também pode resultar em valores modificados.

As regras de promoção são aplicadas automaticamente a expressões que contêm valores de dois ou mais tipos de dados (também chamadas expressões do *tipo misto).* O tipo de cada valor em uma expres-são do tipo misto é promovido automaticamente para o tipo máximo na expressão (na realidade, uma versão temporária de cada valor é criada e usada na expressão — os valores originais permanecem inalterados). A Fig. 5.5 lista os tipos de dados, classificados do maior para o menor, com as especificações de conversão **printf** e **scanf** de cada tipo.

Normalmente, converter valores para tipos inferiores ocasiona resultados incorretos. Portanto, um valor só pode ser convertido para um tipo inferior atribuindo explicitamente o valor a uma variável do tipo inferior ou usando um operador de coerção. Os valores dos argumentos da função são convertidos para os tipos dos parâmetros de seu protótipo se forem atribuídos diretamente a variáveis daqueles tipos.

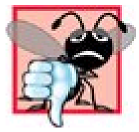
Se nossa função **square** que usa um parâmetro inteiro (Fig. 5.3) fosse chamada com um argumento de ponto flutuante, este seria convertido para **int** (um tipo inferior) e normalmente **square** retornaria um valor incorreto. Por exemplo, **square (4.5)** retornaria **16** e não **20.25.**



Erro comun de programação 5.10

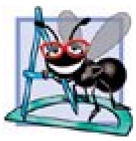
Converter de um tipo superior de dados em uma hierarquia de promoção para um tipo inferior pode modificar o valor dos dados.

Se o protótipo de uma função não for incluído em um programa, o compilador forma seu próprio protótipo de função usando a primeira ocorrência da mesma — ou a definição da função, ou uma chagada para ela. Por default, o compilador assume que a função retorna um **int,** e nada é assumido no diz respeito aos argumentos. Portanto, se os argumentos passados para uma função estiverem incor retos. os erros não serão detectados pelo compilador.



Erro comun de programação 5.11

*Esquecer um protótipo de função causa um erro de sintaxe se o tipo de retorno daquela função não for **int** e sua definição aparecer depois de sua chamada no programa. Caso contrário, esquecer um protótipo de função pode causar um erro em tempo de compilação ou um resultado incorreto.*



Observação de engenharia de software 5.9

Um protótipo de função colocado fora de qualquer definição de função se aplica a todas as chamadas daquela função que aparecem depois de seu protótipo no arquivo. Um protótipo de função colocado dentro de uma função se aplica somente às chamadas realizadas naquela função.

5.7 Arquivos de Cabeçalho

Cada biblioteca padrão tem um *arquivo de cabeçalho* correspondente contendo os protótipos de todas **as fun**ções daquela biblioteca e definições dos vários tipos de dados e constantes necessários por elas. A Fig. 5.6 lista alfabeticamente os arquivos de cabeçalho da biblioteca padrão que podem ser incluídos em programas.

Tipos de dados	Especificações de conversão de printf	Especificações de conversão de scanf
long double	%Lf	%Lf
Double	%f	%lf
Float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
Int	%d	%d
Short	%hd	%hd
Char	%c	%c

Fig. 5.5 Hierarquia de promoção para os tipos de dados.

Arquivos de cabeçalho da biblioteca padrão	Explicação
<assert.h>	Contem macros e informações para adicionar diagnósticos que ajudam o programador a realizar depuração.
<ctype.h>	Contém protótipos de funções que examinam determinadas propriedades dos caracteres e de funções que podem ser usadas para converter letras minúsculas em maiúsculas e vice-versa.
<errno.h>	Define macros úteis para informar condições de erros
<float.h>	Contém limites do sistema para o tamanho dos números de ponto flutuante
<limits.h>	Contém os limites do sistema para os tamanhos dos números inteiros.
<locale.h	Contém protótipos de funções e outras informações que permitem a modificação do programa de acordo com o local onde estiver sendo executado. A noção de local permite ao sistema computacional administrar diferentes convenções para expressar dados como datas, horários, quantidades monetárias e números grandes em diferentes regiões do mundo
<math.h>	Contem protótipos de funções da biblioteca matemática
<setjmp.h>	Contem protótipos de funções que permitem evitar a sequencia normal de chamada e retorno da função
<signal.h	Contem protótipos de funções e macros que administram várias condições que podem surgir durante a execução de um programa.
<stdarg.h>	Define macros que são utilizadas com uma lista de argumentos de uma função, onde o numero e o tipo dos argumentos, são desconhecidos
<stddef.h>	Contém definições comum dos tipos usados pelo C para realizar determinados cálculos.
<stdio.h>	Contém protótipos de funções da biblioteca padrão de entrada/saída e as informações utilizadas por elas.
<stdlib.h>	Contem protótipos de funções para conversão de números em texto e texto em números, alocação de memória, números aleatórios e outras funções com várias finalidades
<string.h>	Contém protótipos de funções para processamento de strings.
<time.h>	Contém protótipos e tipos de funções para manipular horários e datas.

Fig. 5.6 Os arquivos de cabeçalho da biblioteca padrão.

Fig. 5.6 lista alfabeticamente os arquivos de cabeçalho da biblioteca padrão que podem ser incluídos em programas.

O programador pode criar arquivos de cabeçalho personalizados. Os arquivos de cabeçalho defini-dos pelo programador também podem terminar em .**h.** Um arquivo de cabeçalho definido pelo progra-mador pode ser incluído usando a diretiva de pré-processador **#include.** Por exemplo, o arquivo de cabeçalho **square.h** pode ser incluído em nosso programa através da diretiva

#include "square.h"

no topo do programa.

5.8 Chamando Funções: Chamadas por Valor e Chamadas por Referência

As duas maneiras de ativar as funções em muitas linguagens de programação são *chamadas por valor e chamadas por referência.* Quando os argumentos são passados através de uma chamada por valor, é feita uma *cópia* do valor dos argumentos e a mesma é passada para a função chamada. As modificações na cópia não afetam o valor original de uma variável na função que realizou a chamada. Quando um argumento é passado através de uma chamada por referência, a função chamadora permite realmente que a função chamada modifique o valor original da variável.

A chamada por valor deve ser usada sempre que a função chamada não precisar modificar o valor da variável original da função chamadora. Isso evita os *efeitos paralelos (colaterais)* acidentais que retar dam tão freqüentemente o desenvolvimento de sistemas corretos e confiáveis de software. A chamada por referência só deve ser usada com funções confiáveis que precisem modificar a variável original. Em C, todas as chamadas são por valor. Como veremos no Capítulo 7, é possível *simular* a chamada por referência usando operadores de endereços e de ações indiretas. No Capítulo 6, veremos que os arrays são passados automaticamente através de uma simulação de chamada por referência. Será necessário esperar até o Capítulo 7 para se ter um entendimento completo dessa complexa questão. Neste instante, veremos a chamada por valor.

5.9 Geração de Números Aleatórios

Agora vamos mudar para um assunto breve e, esperamos, divertido e tratar de uma aplicação popular de programação, que é a dos jogos e simulações. Nesta seção e na próxima, desenvolveremos um progra-ma de jogo bem-estruturado que inclui várias funções. O programa usa a maioria das estruturas de con-trole que já estudamos.

Há algo no ambiente de um cassino que estimula todos os tipos de pessoas, desde os jogadores pro-fissionais nas mesas de mogno e feltro para o jogo de dados (chamado *craps,* em cassinos) até os peque-nos apostadores em máquinas caça-níqueis. É *o fator sorte,* a possibilidade de que a sorte converta um mero punhado de dinheiro em uma montanha de riqueza. O fator sorte pode ser introduzido em aplica-ções do computador usando a função **rand** da biblioteca padrão do C.
Veja a seguinte instrução:

i = rand ( );

A função **rand** gera um inteiro entre 0 e **RAND_MAX** (uma constante simbólica definida no arquivo de cabeçalho **<stdlib**. h>). O padrão ANSI estabelece que o valor de **RAND_MAX** deve ser menor do **que** 32767, que é o valor máximo de um inteiro de dois bytes (i.e., 16 bits). Os programas nesta seção foram executados em um sistema C com o valor máximo de 32767 para **RAND_MAX.** Se **rand** produ-zir inteiros verdadeiramente aleatórios, todos os números entre 0 e **RAND_MAX** terão a mesma *probabi-lidade* de serem escolhidos cada vez que **rand** for chamada.

```
/* Inteiros em uma escala ajustada e deslocada produzidos por 1 + rand ( ) %6*/
#include <stdio.h>
#include <stdlib.h>
main( )
{
int i ;
for  (i = 1;  i <= 20;  i++) {
printf ("%10d",  1 +  (rand() % 6));
if  (i % 5 == 0)
printf("\n");
}
return 0;
}
```

```
5    5    3    5    5
2    4    2    5    5
5    3    2    2    1
5    1    4    6    4
```

Fig. 5.7 Inteiros em uma escala ajustada e deslocada produzidos por 1 + rand( ) % 6.

Freqüentemente, o conjunto de valores produzido por **rand** é diferente do necessário para uma de terminada aplicação. Por exemplo, um programa que simule o lançamento de uma moeda pode exigir | apenas 0 para "cara" e 1 para "coroa". Um

programa de jogo de dados que simule um dado de seis faces exigiria inteiros aleatórios no intervalo de 1 a 6.

Para demonstrar **rand,** vamos desenvolver um programa que simule 20 lançamentos de um dado de I seis faces e imprimir o valor de cada lançamento. O protótipo da função para **rand** pode ser encontrado em **<stdlib.h>.** Usamos o operador resto (modulus, %) juntamente com **rand** como se segue

rand( ) % 6

para produzir inteiros de 1 a 5. Isso é chamado *ajuste de escala.* O número 6 é chamado *fator de escala.* A seguir, *deslocamos a escala* dos números produzidos adicionando 1 ao nosso resultado anterior. A Fig. 5.7 confirma que os resultados estão contidos no intervalo de 1 a 6.

Para mostrar que esses números ocorrem com probabilidades aproximadamente iguais, vamos simu-lar 6000 lançamentos de um dado com o programa da Fig. 5.8. Cada inteiro de 1 a 6 deve ser obtido 1 aproximadamente 1000 vezes.

Como mostra a saída do programa, fazendo o ajuste e o deslocamento da escala utilizamos a função **rand** para simular realisticamente o lançamento de um dado de seis faces. Observe que *não* é fornecido caso **default** na estrutura **switch.** Observe também o uso do especificador de conversão %s para imprimir as strings de caracteres **"Face"** e **"Freqüência"** como cabeçalhos de colunas. Depois de estudarmos arrays no Capítulo 6, mostraremos como substituir elegantemente toda a estrutura **switch** por uma instrução em uma única linha.

Executar o programa da Fig. 5.7 produz novamente

```
5    5    3    5    5
2    4    2    5    5
5    3    2    2    1
5    1    4    6    4
```

Observe que foi impressa exatamente a mesma seqüência de valores. Como esses números podem ser ale-atórios? Ironicamente, esse repetição de valores é uma característica importante da função **rand.** Ao de-purar um programa, essa repetição é essencial para assegurar que as correções funcionam corretamente.

Na realidade, a função **rand** gera *números pseudo-aleatórios.* Chamar **rand** repetidamente produz uma seqüência de números que parece ser aleatória. Entretanto, a seqüência se repete cada vez que o programa é executado. Depois de o programa ser completamente depurado, ele pode ser condicionado de modo a produzir uma seqüência diferente de números em cada execução. Isso é chamado *randomização,* e é realizado com a função **srand** da biblioteca padrão. A função **srand** utiliza um argumento inteiro **unsigned** para ser a *semente* da função **rand** de forma que seja

produzida uma seqüência diferente de números aleatórios em cada execução do programa.

O uso de **srand** é demonstrado na Fig. 5.9. No programa, usamos o tipo de dado **unsigned** que é uma abreviação de **unsigned int.** Um **int** é armazenado pelo menos nos dois últimos bytes **d** memória e pode ter valores positivos ou negativos. Uma variável do tipo **unsigned** é também arma-zenada em pelo menos dois bytes de memória. Um **unsigned int** de dois bytes só pode ter valores positivos no intervalo de 0 a 65535. Um **unsigned int** de quatro bytes só pode ter valores positivos no intervalo de 0 a 4294967295. A função **srand** utiliza um valor **unsigned** como argumento. O especificador de conversão **%u** é usado para ler um valor **unsigned** com **scanf**. O protótipo da fun-ção **srand** é encontrado em **<stdlib.h>.**

Vamos executar o programa várias vezes e observar os resultados. Observe que uma seqüência *dife-rente* de números aleatórios é obtida cada vez que o programa é executado já que, em cada uma delas, uma semente diferente é fornecida. Se desejássemos randomizar sem a necessidade de fornecer uma semente a cada vez, poderíamos usar uma instrução como

srand(time(NULL));

```
/* 6000 lançamentos de um dado de seis faces */
#include <stdio.h>
#include <stdlib.h>
main( ) {
int face,  jogada,  frequencia1 = 0,  frequencia2 = 0,
                frequencia3 = 0,  frequencla4 = 0,
                freguencia5 = 0,  frequencia6 = 0;
for (jogada = 1,  jogada <= 6000;  jogada++) {
        face = 1 + rand( ) % 6;
        switch (face) {
                case 1:
                        ++frequencia1;
                        break;
                case 2:
                        ++frequencia2;
                        break;
                case 3:
                        ++frequencia3;
                        break;
                case 4:
                        ++frequencia4;
                        break;
                case 5:
                        ++frequencia5;
                        break;
                case 6:
                        ++frequencia6;
                        break;
        }
```

```
}
printf("%s%13s\n", "Face", "Freqüência");
}
```

```
Face         Freqüência
1                 987
2                 984
3                1029
4                 974
5                1004
6                1022
```

Fig. 5.8 Jogando 6000 vezes um dado de seis faces.

```
/* Randomizando o programa de lançamento de um dado */
#include <stdlib.h>
#include <stdio.h>
main() {
int i;
unsigned semente;
printf("Entre com a semente: ");
scanf ("%u",  &Semente) ;
srand(semente);
for (i = 1;  i <= 10;  i++){
        printf ("%10d",   1 +  (rand() % 6));
        if  (i % 5 == 0)
                printf("\n");
}
return 0;
}
```

```
Entre com a semente: 67
1    6    5    1    4
5    6    3    1    2
```

```
Entre com a semente: 432
4    2    5    4    3
2    5    1    4    4
```

```
Entre com a semente: 67
1    6    5    1    4
5    6    3    1    2
```

Fig. 5.9 Randomizando o programa de lançamento de um dado.

Isso faz com que o computador leia seu relógio para obter automaticamente o valor da semente. A função **time** retorna a hora atual do dia em segundos. Esse valor é convertido em um inteiro sem sinal **(unsigned int)** que é usado como semente do gerador de números

aleatórios. A função **time** toma **NULL** como argumento **(time** é capaz de fornecer ao programador uma string representando a hora do dia; **NULL** desativa esta capacidade em uma chamada específica de **time).** O protótipo da função **time** está em **<time.h>.**

Os valores produzidos diretamente por **rand** sempre estão no intervalo:

rand() RAND_MAX

Demonstramos anteriormente como escrever uma instrução simples em C para simular o lançamento de um dado de seis faces:

face = 1 + rand () % 6;

Essa instrução sempre atribui um valor inteiro (aleatoriamente) à variável **face** no intervalo 1 **<= face** <= 6. Observe que a amplitude desse intervalo (i.e., o número de inteiros consecutivos no intervalo) é 6 e o número inicial do intervalo é 1. Examinando a instrução anterior, vemos que a largura do intervalo é determinada pelo número usado para ajustar a escala de **rand** com o operador resto (i.e., 6), e o número inicial do intervalo é igual ao número (i.e., **1)** adicionado a **rand % 6.** Podemos generalizar esse Resultado como se segue

n = a + rand() % b;

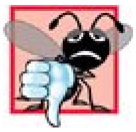
onde **a** é o *valor de deslocamento da escala* (que é igual ao primeiro número do intervalo de inteiros consecutivos desejado) e b é o fator de ajuste de escala (que é igual à amplitude do intervalo de inteiros consecutivos desejado). Nos exercícios, veremos que é possível escolher inteiros aleatoriamente a partir de conjuntos de valores diferentes de inteiros consecutivos.

```
/* Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int rollDice(void);
main() {
int gameStatus,  sum, myPoint;
srand(time(NULL));
sum = rollDice();       /* primeira jogada dos dados */
switch(sum) {
        case 7:
        case 11:        /* vence na primeira jogada */
                gameStatus = 1;
                break;
        case 2:
        case 3:
        case 12:        /* perde na primeira jogada */
                gameStatus = 2;
                break;
        default:        /* armazena o ponto */
                gameStatus = 0;
```

```
                myPoint = sum;
                printf("O ponto e %d\n", myPoint);
                break;
        }
while (gameStatus == 0)   {   /* continua jogando */
        sum = rollDice();
        if(sum == myPoint)    /* vence fazendo o ponto */
                gameStatus = 1;
        else
                if(sum == 7)    /* perde obtendo o valor 7 */
                        gameStatus = 2;
}
if(gameStatus == 1)
        printf("Jogador vence\n") ;
else
        printf("Jogador perde\n");
return 0;
}

int rollDice(void) {
int diel, die2, workSum;
die1 = 1 + (rand() % 6);
die2 = 1 + (rand() % 6);
workSum = die1 + die2;
printf("Jogador obteve %d + %d = %d\n",  die1,  die2, workSum);
return workSum;
}
```

Fig. 5.10 Programa para simular um jogo de dados (craps).



Erro comun de programação 5.12

Usar ***srand*** *no lugar de* ***rand*** *para gerar números aleatórios.*

5.10 Exemplo: Um Jogo de Azar

Um dos jogos de azar mais populares é o jogo de dados conhecido como *"craps"*, que é jogado em cas-sinos e salas de jogos de todo o mundo. As regras do jogo são simples:

Um jogador joga dois dados. Cada dado tem seis faces. Essas faces contêm 1, 2, 3, 4, 5 ou 6 pontos. Depois **4l** *os dados pararem, a soma das pontas nas faces superiores é calculada. Se a soma for 7 ou 11 no primeiro Ian-çamento, o jogador vence. Se a soma for 2, 3 ou 12 no primeiro lançamento (chamado "craps"), o jogador perde (i.e., a "casa" vence). Se a soma for 4, 5. 6, 8. 9 ou 10 no primeiro lançamento, esta soma se torna o "ponto " do jogador. Para vencer, o jogador deve continuar lançando os dados até "alcançar (fazer) seu pon-to ". O jogador perde se tirar um 7 antes de fazer seu ponto.*

O programa da Fig. 5.10 simula o jogo de craps. A Fig. 5.11 mostra várias execuções de exemplo.

Observe que o jogador deve rolar dois dados no primeiro lançamento e deve fazer isso em todos os lançamentos subseqüentes. Definimos a função **rollDice** para rolar os dados e calcular e imprimir a soma dos pontos de suas faces. A função **rollDice** é definida apenas uma vez, mas é chamada em dois locais no programa. É interessante observar que **rollDice** não possui argumentos, portanto indi-camos **void** na lista de parâmetros. A função **rollDice** retorna a soma dos dois dados, portanto um tipo de retorno **int** é indicado no cabeçalho da função.

O jogo é razoavelmente complicado. O jogador pode vencer ou perder no primeiro lançamento ou pode vencer ou perder em qualquer lançamento subseqüente. A variável **gameStatus** é usada para controlar isso.

```
Jogador obteve 6 + 5 = 11 Jogador vence
```

```
Jogador obteve 6 + 6 = 12 Jogador perde
```

```
Jogador obteve 4    + 6    = 10
O ponto e 10
Jogador obteve 2+4=6
Jogador obteve 6    + 5    = 11
Jogador obteve 3+3=6
Jogador obteve 6    + 4    = 10
Jogador vence
```

```
Jogador obteve      1+3=4
O ponto e 4
Jogador obteve      1+4=5
Jogador obteve      5+4=9
```

```
Jogador obteve    4    + 6    = 10
Jogador obteve    6+3=9
Jogador obteve    5+2=7
Jogador perde
```

Fig. 5.11 Exemplos de execuções do jogo de dados (craps).

Quando um jogo é vencido, ou no primeiro lançamento ou em qualquer lançamento subseqüente, **gameStatus** é definida com o valor 1. Quando um jogo é perdido, ou no primeiro lançamento ou em qualquer lançamento subseqüente, **gameStatus** é definida com o valor 2. De outra forma, **gameStatus** É definida com o valor zero e o jogo deve continuar.

Depois do primeiro lançamento, se o jogo terminar, a estrutura **while** é ignorada porque **gameStatus** não será igual a zero. O programa prossegue para a estrutura **if/else** que imprime **"Jogador ven-ce"** se **gameStatus** for igual a 1 e **"Jogador perde"** se **gameStatus** for igual a **2.**

Depois do primeiro lançamento, se o jogo não terminar, a soma **(sum)** é armazenada na variável **myPoint** ("meu ponto"). A execução prossegue com a estrutura **while** porque **gameStatus** é **0.**

Cada vez que a estrutura **while** é percorrida, **rollDice** é chamada para produzir uma nova variável **sum.** Se **sum** for igual a **myPoint, gameStatus** é definido com o valor **1** para indicar que o jogador pilhou, a verificação no **while** fica sendo falsa, a estrutura **if/else** imprime **"Jogador vence"** e a execução termina. Se **sum** for igual a **7, gameStatus** é definido com o valor **2** para indicar e o jogador perdeu, a verificação no **while** fica falsa, a instrução **if/else** imprime **"Jogador perde"** e a execução é encerrada.

Observe a interessante estrutura de controle do programa. Usamos duas funções — **main** e **rollDice** — e as estruturas **switch, while, if/else** e o **if** aninhado. Nos exercícios, examinaremos várias características interessantes do jogo de dados craps.

5.11 Classes de Armazenamento

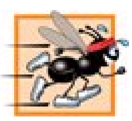
Capítulos 2 a 4, usamos identificadores para nomes de variáveis. Os atributos de variáveis incluem nome, tipo e valor. Neste capítulo, também usamos identificadores como nomes de funções definidas pelo usuário. Na realidade, cada identificador em um programa tem outros atributos, incluindo *classe de armazenamento, tempo (duração) de armazenamento, escopo* e *linkage (ligação).* A linguagem C fornece quatro classes de armazenamento indicadas pelos *especificadores de classes de armazenamento:* **auto, register, extern** e **static.** Uma *classe de armazenamento* de um identificador ajuda a determinar seu tempo de armazenamento, escopo e linkage. Um *tempo de armaze-nento* de um identificador é o período durante o qual aquele identificador permanece na memória. Alguns identificadores permanecem pouco tempo, alguns são criados e eliminados repetidamente, e outros permanecem durante toda a execução do programa. O *escopo* de um identificador é onde se pode fazer referência àquele identificador dentro de um programa. Pode-se fazer referência a alguns identificadores **ao** longo de todo o programa, outros apenas a partir de determinados locais de um programa. A *linkage (ligação)* de um identificador determina, para um programa com vários arquivos-fonte, se um identificador *é* conhecido apenas no arquivo-fonte atual ou em qualquer arquivo-fonte com as declarações adequa-das. Esta seção analisa as quatro classes e o tempo de armazenamento. A Seção 5.12 analisa o escopo dos identificadores.

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento: *tempo de armazenamento automático* e *tempo de armazenamento estático.* As pala-vras-chave **auto** e **register** são utilizadas para declarar variáveis de tempo de armazenamento au-tomático. As variáveis com tempo de armazenamento automático são criadas quando o bloco no qual são declaradas é acionado, elas existem enquanto o bloco estiver ativo e são destruídas quando o bloco *é* deixado para trás.

Apenas variáveis podem ter tempo de armazenamento automático. As variáveis locais de uma função (as declaradas na lista de parâmetros ou no corpo da função) possuem normalmente tempo de armazenamento automático. A palavra-chave **auto** declara explicitamente variáveis de tempo de armazenamento automático. Por exemplo, a declaração a seguir indica que as variáveis **x** e **y** do tipo **float** são variáveis locais automáticas e existem apenas no corpo da função na qual a declaração aparece:

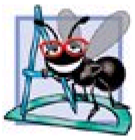
auto float x, y;

As variáveis locais possuem tempo de armazenamento automático por default, portanto a palavra-chave **auto** raramente é usada. Para o restante do texto, vamos nos referir a variáveis com tempo de armazenamento automático apenas como variáveis automáticas.



Dica de desempenho 5.1

O armazenamento automático é um meio de economizar memória porque as variáveis automáticas só existem quando são necessárias. Elas são criadas quando é acionada a função na qual são declaradas, e são destruídas quando a função é encerrada.



Observação de engenharia de software 5.10

O armazenamento automático é ainda mais um exemplo do princípio do privilégio mínimo. Por que armazenar variáveis na memória e torná-las acessíveis quando na realidade não são mais necessárias ?

Os dados na versão de linguagem de máquina de um programa são carregados normalmente em registros para cálculos e outros processamentos.



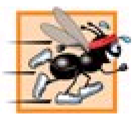
Dica de desempenho 5.2

*O especificador **register** de classe de armazenamento pode ser colocado antes de uma declaração de variável automática para indicar ao compilador que a variável seja conservada em um dos registradores de hardware de alta velocidade do computador. Se as variáveis forem usadas freqüentemente como contadores ou para cálculo de totais, elas podem ser conservadas em registros de hardware, o overhead de carregar repetidamente as variáveis da memória nos registros e armazenar os resultados novamente na memória pode ser eliminado.*

O compilador pode ignorar as declarações **register.** Por exemplo, pode não haver número suficien-te de registros disponíveis para o uso do compilador. A declaração a seguir indica que a variável inteira **I contador** pode ser colocada em um dos registros do computador e inicializada como 1:

register int contador = 1;

A palavra-chave **register** só pode ser usada com variáveis de tempo de armazenamento automático.

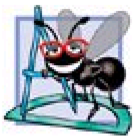


Dica de desempenho 5.3

*Freqüentemente, as declarações **register** são desnecessárias. Os compiladores otimizados de hoje são capazes de reconhecer as variáveis usadas repetidamente e podem decidir colocá-las em registros sem que haja a necessidade de o programador incluir uma declaração **register.***

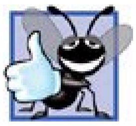
As palavras-chave **extern** e **static** são usadas para declarar identificadores para variáveis e fun-ções de tempo de armazenamento estático. Os identificadores de tempo de armazenamento estático j existem desde o instante em que o programa começou a ser executado. Para variáveis, o armazenamento é alocado e inicializado uma vez quando o programa começa a ser executado. Para funções, o nome da função existe quando o programa começa a ser executado. Entretanto, muito embora as variáveis e os nomes das funções existam quando o programa começa a ser executado, isso não significa que esses I identificadores podem ser usados em todo o programa. O tempo de armazenamento e o escopo (onde uni nome pode ser usado) são questões diferentes, como veremos na Seção 5.12.

Há dois tipos de identificadores com tempo de armazenamento estático: identificadores externos (como variáveis globais e nomes de funções) e variáveis locais declaradas com o especificador de classe de arma-zcnamento **static.** As variáveis globais e nomes de funções são pertencentes à classe de armazenamen-to **extern** por default. As variáveis globais são criadas colocando declarações de variáveis fora de qual-quer definição de função, e conservam seus valores ao longo de toda a execução do programa. As referên-cias a variáveis globais e funções podem ser feitas em qualquer função que venha após suas declarações ou definições no arquivo. Essa é uma razão para o uso de protótipos de funções. Quando incluímos **stdio.h** em um programa que utiliza **printf,** o protótipo da função é colocado no início de nosso arquivo para | fazer com que o nome **printf** seja conhecido para todo o restante do arquivo.



Observação de engenharia de software 5.11

Declarar uma variável como global em vez de local permite a ocorrência de efeitos colaterais indesejáveis quando uma função modifica acidental ou intencionalmente uma variável, à qual não precisa ter acesso. Em geral, o uso de variáveis globais deve ser evitado, exceto em determinadas situações com exigências especiais de desempenho.



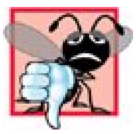
Boa prática de programação 5.10

As variáveis usadas apenas em uma determinada função devem ser declaradas como variáveis locais naquela função em vez de variáveis externas.

As variáveis locais declaradas com a palavra-chave **static** são conhecidas apenas na função na qual são definidas, mas, diferentemente das variáveis automáticas, as variáveis locais **static** conser-vam seus valores quando a função é encerrada. Na próxima vez em que a função for chamada, a variável local **static** conservará o valor que tinha quando a função foi executada pela última vez. A instrução seguinte declara a variável local **contagem** como **static** e a inicializa com o valor 1.

```
static int contagem =1;
```

Todas as variáveis numéricas de tempo de armazenamento estático são inicializadas com o valor zero se não forem inicializadas explicitamente pelo programador. (As variáveis denominadas ponteiros, anali-sadas no Capítulo 7, são inicializadas com o valor NULL.)



Erro comun de programação 5.13

Usar vários especificadores de classe de armazenamento para um identificador. Apenas um especificador de classe de armazenamento pode ser aplicado a um identificador.

As palavras-chave **extern e static** possuem significado especial quando aplicadas explicitamente a identificadores externos.

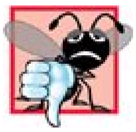
5.12 Regras de Escopo

Um *escopo* de um identificador é a parte do programa na qual ele pode ser referenciado. Por exemplo, quando declaramos uma variável local em um bloco, podem ser feitas referências a ela apenas naquele bloco ou em blocos ali aninhados. Os quatro escopos de um identificador são *escopo de função, escopoarquivo, escopo de bloco* e *escopo de protótipo de função.*

Os rótulos (ou *labels*, que são identificadores seguidos de dois pontos, como **start** :) são os únicos identificadores com *escopo de função.* Os rótulos podem ser usados em qualquer lugar de uma função na qual aparecem, mas não pode ser feita qualquer referência a eles fora do corpo da função. Os labels são usados em estruturas **switch** (como os labels **case)** e em instruções **goto.** Os labels são detalhes de implementação que funções ocultam de outras funções. Essa ocultação — chamada mais formalmente de *ocultação de informações* — é um dos princípios mais importantes da boa engenharia de software. Um identificador declarado fora de qualquer função tem *escopo de arquivo.* Tal identificador é "co-nhecido" por todas as funções desde o local onde é declarado até o final do arquivo. Variáveis globais, definições de funções e protótipos de funções colocados fora de uma função possuem escopo de arquivo.

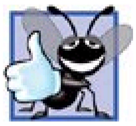
Os identificadores declarados dentro de um bloco possuem *escopo de bloco.* O escopo de bloco termina na chave direita final (}) do bloco. As variáveis locais declaradas no início de uma função possuem escopo de bloco, assim como os parâmetros da função, que são considerados variáveis locais por ida. Qualquer bloco pode conter declarações de variáveis. Quando os blocos estão aninhados e um identificador em um bloco externo tem o mesmo nome de um identificador em um bloco interno, o identificador no bloco externo é "escondido" até que o bloco interno seja encerrado. Isso significa que durante a execução de um bloco interno, este vê o valor de seu próprio identificador local, e não o valor do identificador de mesmo nome no bloco externo. As variáveis locais declaradas **static** também possuem escopo de bloco, muito embora existam desde o instante em que o programa começou a ser executado. Dessa maneira, o tempo de armazenamento não afeta o escopo de um identificador.

Os únicos identificadores com *escopo de protótipo de função* são os utilizados na lista de parâmetros de um protótipo de função. Como já mencionamos, os protótipos de funções não necessitam de nomes em suas listas de parâmetros — apenas os tipos são exigidos. O nome que for usado na lista de parâmetros de um protótipo de função será ignorado pelo compilador. Os identificadores usados em um protótipo de função podem ser reutilizados em qualquer lugar do programa sem ambigüidade.



Erro comun de programação 5.14

Usar acidentalmente o mesmo nome já usado em um bloco externo para um identificador em um bloco inter-no, quando, na realidade, o programador desejava que o identificador do bloco externo estivesse ativo durante o processamento do bloco interno.



Boa prática de programação 5.11

Evite nomes de variáveis que desativem (escondam) nomes em escopos externos. Isso pode ser realizado simplesmente evitando o uso de identificadores repetidos em um programa.

O programa da Fig. 5.12 demonstra as características dos escopos com variáveis globais, variáveis locais automáticas e variáveis locais static. Uma variável global *x é* declarada e inicializada com o valor 1. Essa variável global é ignorada em qualquer bloco (ou função) que declare uma variável x. *t* main, a variável local x é declarada e inicializada com o valor 5. A seguir, essa variável é impressa para mostrar que a variável global x é ignorada em main. Em seguida, é definido um novo bloco em main com outra variável local x inicializada com o valor 7. Essa variável é impressa para mostrar que x do bloco externo de main é ignorada. A variável x com o valor 7 é eliminada automaticamente depois da execução do bloco, e a variável local *x* do bloco externo de main é impressa novamente para mostrar que não está mais sendo ignorada. O programa define três funções que não utilizam nenhum argumento e não retornam nenhum valor. A função a define uma variável automática *x* e a inicializa como 25. Quando a é chamada, a variável é impressa, incrementada e impressa novamente antes do término da função. Cada vez que essa função é chamada, a variável automática x é reinicializada como 25. A fun-ção b declara uma variável (static) x e a inicializa como 50. As variáveis locais declaradas como static conservam seus valores mesmo quando estão fora do escopo. Quando b é chamada, a variável *x* é impressa, incrementada e impressa novamente antes do término da função. Na próxima chamada essa função, a variável static local x terá o valor 51. A função c não declara variável alguma. Por-tanto, quando ela faz referência à variável x, o *x* global é utilizado. Quando c é chamada, a variável global é impressa, multiplicada por 10 e impressa novamente antes do término da função.

Na próxima I vez que a função c for chamada, a variável global ainda possuirá o valor modificado, 10. Finalmente. o programa imprime mais uma vez a variável local x em main para mostrar que nenhuma das chamadas das funções modificou o valor de x porque todas as funções utilizavam variáveis de seus escopos.

5.13 Recursão

Geralmente, os programas que analisamos são estruturados como funções que fazem chamadas entre si de uma maneira disciplinada e hierárquica. Para alguns tipos de problemas, é útil ter funções que chamem a si mesmas. Uma *função recursiva* é uma função que chama a si mesma, ou diretamente ou indiretamente por meio de outra função. A recursão é um tópico complexo analisado exaustivamente em cursos superiores de-ciência da computação. Nesta seção e na seção seguinte, são apresentados exemplos simples de recursão Este livro contém um tratamento amplo de recursão que está distribuído ao longo dos Capítulos 5 a 12. A Fig. 5.17, colocada no final da Seção 5.15, resume os 31 exemplos de recursão e exercícios do livro. Em primeiro lugar, vamos analisar o conceito de recursão e depois examinaremos vários programas contendo funções recursivas. Os métodos recursivos para solução de problemas possuem vários elementos em comum. Uma função recursiva é chamada para resolver o problema. Na realidade, a função só sabe resolver o(s) caso(s) mais simples, ou o(s) chamado(s) *caso(s) básico(s).* Se a função for chamada *em* um problema mais complexo, ela divide o problema em duas partes teóricas: uma parte que a função sabe como resolver e outra que ela não sabe.
Para tornar viável a recursão, a segunda parte deve ser parecida com o problema original, mas ser uma versão um pouco mais simples ou menor do que ele. Por esse novo problema ser parecido com o problema original, a função chama (lança) uma nova cópia de si mesma para lidar com o problema menor — o que é conhecido por *chamada recursiva* ou *etapa de recursão.* A etapa de recursão também inclui a palavra-chave return porque seu resultado será com binado com a parte do problema que a função sabe como resolver para formar um resultado que será enviado de volta para a função original de chamada, possivelmente **main.**

```
/* Um exemplo de escopos */
#include <stdio.h>
void a(void);   /* protótipo de funcao */
void b(void);   /* protótipo de funcao */
void c(void);   /* protótipo de funcao */
int x = 1;      /* variável global */
main() {
int x = 5;      /* variável local para main */
printf("x local no escopo externo de main e %d\n", x) ;
{       /* inicia novo escopo */
        int x = 7;
        printf("x local no escopo interno de main e %d\n", x) ;
}       /* encerra novo escopo */
printf("x local no escopo externo de main e %d\n", x) ;
a();    /* a tem x local automático */
b();    /* b tem x local estático */
c() ;   /* c usa x global */
a();   /* a reinicializa x local automático */
b();    /* x local estático conserva seu valor anterior */
c();    /* x global também conserva seu valor */
printf("x local em main e %d\n", x); return 0;
}
void a(void) {
int x = 25;     /* inicializada sempre que a e chamada */
```

```
printf("\nx local em a e %d depois de entrar em a\n", x);
++x;
printf("x local em a e %d antes de sair de a\n", x);
}
void b(void) {
static int x = 50;        /* apenas inicialização estática */
/* primeira vez que b e chamada */
printf("\nx local estático e %d ao entrar em b\n", x);
++x;
printf("x local estático e %d ao sair de b\n", x);
void c(void) {
printf("\nx global e %d ao entrar em c\n", x);
x *= 10;
printf("x global e %d ao sair de c\n", x);
}
```

```
x local no escopo externo de main e 5
x local no escopo externo de main e 7
x local no escopo externo de main e 5
x local em a e 25 depois de entrar em a
x local em a e 26 antes de sair de a
x local estático e 50 ao entrar em b
x local estático e 51 ao sair de b
x global e 1 ao entrar em c
x global e 10 ao sair de c
x local em a e 25 depois de entrar em a
x local em a e 26 antes de sair de a
x local estático e 51 ao entrar em b
x local estático e 52 ao sair de b
x global e 10 ao entrar em c
x global e 100 ao sair de c
x local em main e 5
```

Fig. 5.12 Um exemplo de escopos.

A etapa de recursão é executada enquanto a chamada original para a função estiver ativa, i.e., ainda não tiver sido concluída. A etapa de recursão pode levar a outras chamadas recursivas, à medida que a função continuar a dividir cada problema em duas partes conceituais. Para a recursão chegar ao fim, cada vez que a função chamar a si mesma com uma versão ligeiramente mais simples do problema original, essa seqüência de problemas cada vez menores deve convergir posteriormente para o caso básico. Nesse instante, a função reconhece o caso básico, envia um resultado de volta para a cópia anterior da função e ocorre uma seqüência de remessa de resultados na ordem inversa até a chamada original da função enviar mais tarde o resultado final para **main.** Tudo isso parece um tanto esquisito, se comparado com a maneira convencional de resolver problemas que usamos até aqui. Na verdade, é necessário muita prática com programas recursivos antes de o processo parecer natural. Como exemplo da utilização desses conceitos, vamos escrever um programa recursivo para realizar um conhecido cálculo da matemática.

O fatorial de um inteiro não-negativo a, escrito ***n!*** (e pronunciado "fatorial de n"), é o produto

$$n * (n - 1) * (n - 2) * ... * 1$$

com 1! igual a **1** e definindo **0!** igual a **1.** Por exemplo, **5!** é o produto **5*4*3*2*1,** que é igual a **120** O fatorial de um inteiro, **numero,** maior ou igual a **0,** pode ser calculado *iterativamente* (não-recursivamente) usando **for** como se segue:

```
fatorial = 1;
for  (contador = numero;  contador >= 1; contador--)
fatorial *= contador;
```

Chega-se a uma definição recursiva para a função fatorial observando o seguinte relacionamento:

*n! = n * (n - 1)!*

Por exemplo, **5**! é claramente igual a **5 * 4!** como é mostrado a seguir:
5! = 5 ** 4 * 3 * 2 * 1 5! = 5 * (4 * 3 * 2 * 1) 5! = 5 * (4!)*

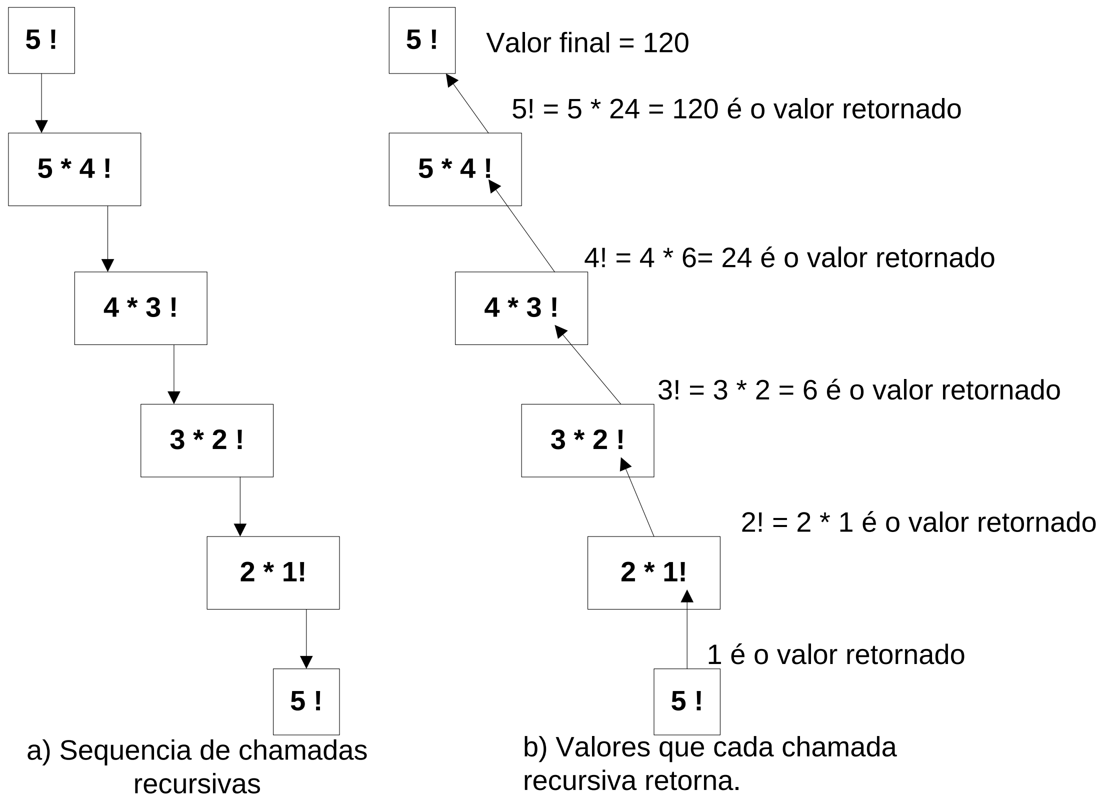


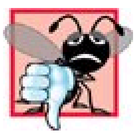
Fig. 5.13 Cálculo recursivo de 5!.

O cálculo de 5! prosseguiria da forma mostrada na Fig. 5.13. A Fig. 5.13a mostra como a sucessão de chamadas recursivas se processaria até 1! ser calculado como 1, que encerraria a recursão. A Fig. 5.13b mostra os valores enviados de volta à função chamadora em cada chamada recursiva até que o valor final seja calculado e retornado. O programa da Fig. 5.14 usa a recursão para calcular e imprimir fatoriais de inteiros de 0 a 10 (a escolha do tipo de dado **long** será explicada em breve). A função recursiva **fatorial** verifica inicialnente se uma condição de término é verdadeira, i.e., se **numero** é menor ou igual a 1. Se **numero** for menor ou igual a 1, **fatorial** retorna 1, nenhuma recursão se faz mais necessária e o programa é encerrado. Se **numero** for maior do que 1, a instrução

return numero * fatorial (numero — 1);

expressa o problema como o produto de **numero** por uma chamada recursiva a **fatorial** calculando o fatorial de **numero — 1.** Observe que **fatorial (numero — 1**) é um problema ligeiramente mais simples do que o cálculo original **fatorial (numero).**

A função **fatorial** foi declarada de modo a receber um parâmetro do tipo **long** e retornar um resultado do tipo **long.** Isso é uma abreviação de **long int.** O padrão ANSI especifica que uma variável do tipo **long int** é armazenada em pelo menos 4 bytes, podendo assim conter o valor máximo de + 2147483647. Como pode ser observado na Fig. 5.14, os valores dos fatoriais se tornam grandes rapidamente. Escolhemos um tipo de dado **long** para que o programa possa calcular fatoriais maiores do 7! em computadores com inteiros pequenos (como 2 bytes). O especificador de conversão **%ld** é utilizado para imprimir valores **long.** Infelizmente, a função fatorial produz valores grandes tão rapidamente que até mesmo a **long int** não permite imprimir muitos valores de fatoriais antes de o tamanho de uma variável **long int** ser superado. Como iremos explorar nos exercícios, **float** e **double** podem ser necessários em último caso para o usuário que deseje calcular fatoriais de números grandes. Isso indica uma deficiência do C (e da maioria das linguagens de programação), ou seja, a de que a linguagem não é ampliada facilmente para atender às exigências especiais de várias aplicações. O C++ é uma linguagem extensível que permite a criação arbitrária de inteiros grandes se assim desejarmos.



Erro comun de programação 5.15

Esquecer-se de retornar um valor de uma função recursiva quando é necessário.

```
/* Funcao recursiva para fatoriais */
#include <stdio.h>
long fatorial(long) ;
main() {
int i ;
for  (i = 1;   i <= 10; i++)
        printf("%2d!  = %ld\n",  i,  fatorial(i));
return 0;
}
/* Definição recursiva da funcao fatorial */
long fatorial(long numero)
{
if  (numero <= 1)
        return 1;
else
        return (numero * fatorial(numero - 1));
}
```

```
1!  =  1
2!  =  2
3!  =  6
4!  =  24
5!  =  120
6!  =  720
7!  =  5040
8!  =  40320
9!  =  362880
10! =  3628800
```

Capítulo	Exemplos e Exercícios de Recursão
Capítulo 5	Função fatorial
	Função de Fibonacci
	Maior divisor comum
	Soma de dois inteiros
	Multiplicação de dois inteiros
	Elevar um inteiro a uma potencia inteira
	Torres de Hanói
	Main recursivo
	Imprimir entradas do teclado na ordem inversa
	Visualizar recursão
Capitulo 6	Soma dos elementos de uma matriz
	Imprimir uma matriz
	Imprimir uma matriz na ordem inversa
	Imprimir uma string na ordem inversa
	Verificar se uma string é um palíndromo
	Valor mínimo de uma matriz
	Ordenação seletiva
	Ordenação rápida
	Pesquisa binária
	Eight Queens ("oito damas")
Capitulo 7	Percorrer um labirinto
Capitulo 8	Imprimir na ordem inversa uma entrada de string pelo teclado
Capítulo 12	Inserção em lista encadeada
	Eliminação em lista encadeada
	Pesquisa em lista encadeada
	Imprimir uma lista encadeada na ordem inversa
	Inserção em árvore binária
	Percorrer uma árvore binária no modo *preorder*
	Percorrer uma árvore binária no modo *inorder*
	Percorrer uma árvore binária no modo *postorder*

Fig. 5.17 Resumo dos exemplos e exercícios sobre recursão no texto.

Resumo

• A melhor maneira de desenvolver e fazer a manutenção de um programa grande é dividi-lo em vários módulos de programas menores, sendo cada um deles mais maleáveis do que o programa original. Os módulos são escritos como funções em C.
• Uma função é ativada por uma chamada de função. A chamada de uma função menciona a função pelo nome e fornece informações (como argumentos) de que a função chamada precisa para realizar sua tarefa.
• O objetivo de ocultar informações é o de permitir o acesso das funções apenas às informações de que elas precisam para concluir suas tarefas. Isso é um meio de implementar o princípio de privilégio mínimo, um dos princípios mais importantes da boa engenharia de software.
• Normalmente as funções são ativadas em um programa escrevendo o nome da função seguido de um parêntese esquerdo, do *argumento* (ou uma lista de argumentos separados por vírgulas) e de um parêntese direito.
• O tipo de dado **double** é um tipo de ponto flutuante como o **float.** Uma variável do tipo **double** pode armazenar um valor de grandeza e precisão muito maiores do que uma do tipo **float.**
• Cada argumento de uma função pode ser uma constante, uma variável ou uma expressão.
• Uma variável local é conhecida apenas em uma definição de função. Outras funções não podem tomar conhecimento do nome das variáveis locais de uma determinada função, nem é permitido a uma função tomar conhecimento dos detalhes de implementação de qualquer outra função.
• O formato geral da definição de uma função é
tipo-do-valor-de-retorno nome-da-função (lista-de-parâmetros) {
declarações instruções
}
O *tipo-do-valor-de-retorno* indica o tipo do valor enviado de volta para a função que fez a chamada. Se uma função não retornar um valor, o *tipo-do-valor-de-retorno* é declarado como **void.** O *nome-da-função é* qualquer identificador válido. A *lista-de-parâmetros* é uma lista separada por vírgulas que contém as declarações das variáveis que serão passadas à função. Se uma função não receber valor algum, a *lista-de-parâmetros é* declarada **void.** O *corpo-da-função* é o conjunto de declarações e instruções que formam a função.
• Os argumentos passados a uma função devem ser equivalentes em número, tipo e ordem aos parâmetros na definição da função.
• Quando um programa encontra uma função, o controle é transferido do ponto de chamada para a referida função, as instruções da função chamada são executadas e o controle retorna a quem chamou.
• Uma função chamada pode retornar de três maneiras o controle a quem chamou. Se a função não retorna um valor, o controle é retornado quando a chave direita final da função é alcançado ou executando a instrução

return;

Se a função retornar um valor, a instrução **return expressão;**
retorna o valor de **expressao.**
• Um protótipo de função declara o tipo de retorno da função e declara o número, os tipos e a ordem dos parâmetros que a função espera receber.
• Os protótipos de funções permitem que o compilador verifique se as funções são chamadas corretamente,
• O compilador ignora os nomes de variáveis mencionados no protótipo da função.

• Cada biblioteca padrão tem um arquivo de cabeçalho correspondente contendo os protótipos de todas as funções naquela biblioteca, assim como as definições das várias constantes simbólicas de que aquelas funções necessitam.
• Os programadores podem criar e incluir seus próprios arquivos de cabeçalho.
• Quando um argumento é passado através de uma chamada por valor, é feita uma *cópia* do valor da variável e esta é passada para a função chamada. As modificações na cópia, no interior da função chamada, não afetam o valor original da variável.
• Todas as chamadas em C são chamadas por valor.
• A função **rand** gera um inteiro entre 0 e **RAND_MAX** que tem o valor máximo definido pelo padrão ANSI C como 32767.
• Os protótipos das funções **rand** e **srand** estão contidos na biblioteca **<stdlib.h>.**
• Os valores produzidos por **rand** podem ter sua escala ajustada e deslocada, de forma a produzir valores em um intervalo específico.
• Para randomizar um programa, use a função **srand** da biblioteca padrão do C.
• Normalmente a instrução **srand** é inserida em um programa depois que este foi completamente depurado. Durante a depuração, é melhor omitir **srand.** Isso assegura a repetição de valores, que é fundamental para provar que as correções em um programa de geração de números aleatórios funcionam adequadamente.
• Para randomizar sem a necessidade de fornecer uma semente para cada execução, podemos usar **srand (time (NULL))**. A função **time** retorna o número de segundos desde o início do dia. **O** protótipo da função **time** está localizado no cabeçalho **<time.h>.**
• A equação geral para ajuste e deslocamento da escala de um número aleatório é

n = a + rand() % b;

onde **a** é o valor de deslocamento (que é igual ao primeiro número do intervalo de inteiros consecutivos desejado) e **b** é o fator de escala (que é igual à amplitude do intervalo de inteiros consecutivos desejado).
• Cada identificador de um programa tem os atributos de classe de armazenamento, tempo (duração) de armazenamento, escopo e linkage (ligação).
• A linguagem C fornece quatro classes de armazenamento indicadas pelos especificadores de classes de armazenamento: **auto, register, extern** e **static.**
• O tempo de armazenamento de um identificador é o período em que ele fica na memória.
• O escopo de um identificador é onde o identificador pode ser mencionado em um programa.
• A ligação de um identificador determina, em um programa de vários arquivos-fonte, se um identificador é conhecido apenas no arquivo-fonte atual ou em qualquer arquivo-fonte, com as declarações adequadas.
• As variáveis com tempo de armazenamento automático são criadas quando se ingressa no bloco onde elas são declaradas, existem enquanto o bloco estiver ativo e são destruídas quando o bloco chega ao fim. Normalmente, as variáveis locais de uma função possuem tempo automático de armazenamento.
• O especificador de classe de armazenamento **register** pode ser colocado antes de uma declaração de variável automática para sugerir que o compilador mantenha a variável em um dos registros de hardware de alta velocidade do computador. O compilador pode ignorar declarações **register.** A palavra-chave **register** pode ser usada apenas com variáveis de tempo automático de duração.
• As palavras-chave **extern** e **static** são usadas para declarar identificadores para variáveis e funções de tempo estático de armazenamento.
• As variáveis com tempo estático de armazenamento são alocadas e inicializadas apenas uma vez quando a execução do programa é iniciada.

• Há dois tipos de identificadores com tempo estático de armazenamento: identificadores externos (como variáveis globais e nomes de funções) e variáveis locais declaradas com o especificador de classe de armazenamento **static.**
• As variáveis globais são criadas colocando declarações de variáveis fora de qualquer definição de função, e conservam seus valores ao longo de toda a execução do programa.
• As variáveis locais declaradas como **static** conservam seus valores quando termina a função na qual são declaradas.
• Todas as variáveis numéricas de tempo estático de armazenamento são inicializadas com o valor zero se não forem inicializadas explicitamente pelo programador.
• Os quatro escopos de um identificador são escopo de função, escopo de arquivo, escopo de bloco e escopo de protótipo de função.
• Os rótulos (labels) são os únicos identificadores com escopo de função. Os rótulos podem ser usados em qualquer lugar da função onde aparecem, mas não se pode fazer referência a eles fora do corpo da função.
• Um identificador declarado fora de qualquer função tem escopo de arquivo. Tal identificador é "conhecido" por todas as funções desde o ponto onde o identificador é declarado até o final do arquivo.
• Os identificadores declarados dentro de um bloco possuem escopo de bloco. O escopo de bloco termina na chave direita (}) de encerramento do bloco.
• As variáveis locais declaradas no início de uma função possuem escopo de bloco como os parâmetros da função, que são considerados variáveis locais por ela.
• Qualquer bloco pode conter declarações de variáveis. Quando houver blocos aninhados, e um identificador externo ao bloco tiver o mesmo nome que um identificador interno a ele, o identificador do bloco externo é "ignorado" ("escondido") até que o bloco interno termine.
• Os únicos identificadores com escopo de protótipo de função são os utilizados na lista de parâmetros de um protótipo de função. Os identificadores empregados em um protótipo de função podem ser reutilizados em qualquer lugar do programa sem que ocorra ambigüidade.
• Uma função recursiva é uma função que chama a si mesma, direta ou indiretamente.
• Se uma função recursiva for chamada com um caso básico, ela simplesmente retorna um resultado. Se a função for chamada com um problema mais complexo, ela divide o problema em duas partes teóricas: uma parte com a qual a função sabe como proceder e uma versão ligeiramente menor do problema original. Como esse novo problema se parece com o problema original, a função ativa uma chamada recursiva para trabalhar com o problema menor.
• Para uma recursão terminar, cada vez que uma função recursiva chamar a si mesma com uma versão ligeiramente mais simples do problema original, a seqüência de problemas cada vez menores deve convergir para o caso básico. Quando a função reconhecer o caso básico, o resultado é enviado para a chamada da função anterior e acontece uma seqüência de remessa de valores retorno até a chamada original da função retornar o resultado final.
• O padrão ANSI não especifica a ordem na qual os operandos da maioria dos operadores (incluindo +) devem ser calculados. Dos muitos operadores do C, o padrão especifica a ordem de cálculo dos operandos dos operadores &&, | |, o operador vírgula (,) e ?: . Os três primeiros são operadores binários cujos dois operandos são calculados da esquerda para a direita. O último operador é o único operador ternário do C. Seu operando da extremidade esquerda é avaliado em primeiro lugar; se o operando mais à esquerda tiver valor diferente de zero, o operando do meio é avaliado em seguida e o último operando é ignorado; se o operando mais à esquerda tiver o valor zero, o terceiro operando é avaliado a seguir e o operando do meio é ignorado.
• Tanto a iteração como a recursão se baseiam em uma estrutura de controle: a iteração usa uma estrutura de repetição; a recursão usa uma estrutura de seleção.

- Tanto a iteração como a recursão envolvem repetições: a iteração usa explicitamente uma estrutura de repetição; a recursão consegue repetição através de chamadas repetidas de funções.
- A interação e a recursão envolvem testes de encerramento (terminação): a iteração se encerra quando a condição de continuação do loop se tornar falsa; a recursão termina quando um caso básico é reconhecido.
- A iteração e a recursão podem ocorrer indefinidamente: acontece um loop infinito com iteração se o teste de continuação do loop nunca se tornar falso; acontece recursão infinita se a etapa de recursão não reduzir o problema de forma que ele convirja para o caso básico.
- A recursão ativa repetidamente o mecanismo, e conseqüentemente o overhead, de chamadas de funções. Isso pode custar caro tanto em termos de tempo do processador como em espaço de memória.

Terminologia

abstração ajuste de escala
argumento em uma chamada de função
armazenamento automático arquivo de cabeçalho
arquivos de cabeçalho da biblioteca padrão
biblioteca padrão do C
bloco
caso básico na recursão chamada por referência chamada por valor chamada recursiva chamadora chamar uma função
classes de armazenamento **clock**
coerção de argumentos compilador otimizado cópia de um valor declaração de função definição de função
deslocamento de escala dividir e conquistar duração do armazenamento
efeitos colaterais efeitos secundários
engenharia de software escopo
escopo de arquivo
escopo de bloco
escopo de função
escopo de protótipo de função
especificador **auto** de classe de armazenamento
especificador de classe de armazenamento
especificador de classe de armazenamento **register**
especificador de classe de armazenamento **static**
especificador de conversão ***%s***
expressões do tipo misto
fator sorte função
função chamada
função definida pelo programador
função recursiva
funções da biblioteca matemática
geração de números aleatórios
hierarquia de promoção
invocar uma função
iteração
ligação
linkage
números pseudo-aleatórios
ocultação de informações
parâmetro em uma definição de função
princípio do privilégio mínimo
programa modular
protótipo de função
rand
RAND_MAX
randomizar
recursão
return
reutilização de software
simulação
srand
tempo automático de armazenamento tempo de armazenamento tempo de armazenamento estático **time**
tipo-de-valor-de-retorno
unsigned variável automática variável global variável local variável **static**
void

Erros Comuns de Programação

5.1 Esquecer de incluir o arquivo de cabeçalho matemático ao usar funções da biblioteca matemática pode levar a resultados errados.

5.2 Omitir o tipo-do-valor-de-retorno em uma definição de função causa um erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int.**

5.3 Esquecer de retornar um valor de uma função que deve fazer isso pode levar a erros inesperados. O padrão ANSI estabelece que o resultado dessa omissão não é definido.

5.4 Retornar um valor de uma função cujo tipo de retorno foi declarado **void** causa um erro de sintaxe. **5.5** Declarar parâmetros da função do mesmo tipo como **float x, y** em vez de **float x, float y.** A declaração de parâmetros **float x, y** tornaria na realidade **y** um parâmetro do tipo **int** porque **int** é o default.

5.5 Colocar um ponto-e-vírgula após o parêntese direito ao encerrar a lista de parâmetros de uma definição de função é um erro de sintaxe. **5.7** Definir um parâmetro de função novamente como variável local dentro da função é um erro de sintaxe.

5.6 Definir uma função dentro de outra função é um erro de sintaxe.

5.7 Esquecer o ponto-e-vírgula no final de um protótipo de função causa um erro de sintaxe.

5.8 Converter de um tipo superior de dados em uma hierarquia de promoção para um tipo inferior pode modificar o valor dos dados.

5.9 Esquecer um protótipo de função causa um erro de sintaxe se o tipo de retorno daquela função não for **int** e sua definição aparecer depois de sua chamada no programa. Caso contrário, esquecer um protótipo de função pode causar um erro em tempo de compilação ou um resultado inesperado.

5.10 Usar **srand** no lugar de **rand** para gerar números aleatórios.

5.11 Usar vários especificadores de classe de armazenamento para um identificador. Apenas um especificador de classe de armazenamento pode ser aplicado a um identificador.

5.12 Usar acidentalmente o mesmo nome já usado em um bloco externo para um identificador em um bloco interno, quando, na realidade, o programador desejava que o identificador do bloco externo estivesse ativo durante o processamento do bloco interno.

5.13 Esquecer-se de retornar um valor de uma função recursiva quando isso é necessário.

5.14 Omitir o caso básico, ou escrever incorretamente a etapa de recursão de forma que ela não convirja para o caso básico, ocasionará uma recursão infinita, fazendo com que finalmente a memória se esgote. Isso é análogo ao problema de um loop infinito em uma solução iterativa (não-recursiva). A recursão infinita também pode ser causada pelo fornecimento de um dado incorreto de entrada.

5.15 Escrever programas que dependam da ordem de cálculo dos operandos de operadores diferentes de &&, | |, ?: e o operador vírgula (,) pode causar erros porque os compiladores podem não calcular os operandos necessariamente na ordem em que o programador espera.

5.16 Fazer acidentalmente uma chamada não-recursiva de uma função para si mesma, tanto diretamente como através de outra função.

Práticas Recomendáveis de Programação

5.1 Familiarize-se com o ótimo conjunto de funções da biblioteca padrão do ANSI C.

5.2 Inclua o arquivo de cabeçalho (header) matemático usando a diretiva do pré-processador **#include <math.h>** ao usar funções da biblioteca matemática.

5.3 Coloque uma linha em branco entre as definições das funções para separá-las e melhorar a legibilidade do programa.

5.4 Embora a omissão de um tipo de retorno seja assumido como **int,** sempre declare explicitamente o tipo de retorno. Entretanto, o tipo de retorno para **main** é normalmente omitido.

5.5 Inclua o tipo de cada parâmetro na lista de parâmetros, mesmo que aquele parâmetro seja do tipo default **int.**

5.6 Embora não seja incorreto fazer isso, não use os mesmos nomes para os argumentos passados para uma função e os parâmetros correspondentes na definição da função. Isso ajuda a evitar ambigüidade.

5.7 Escolher nomes de função e parâmetros significativos torna os programas mais legíveis e ajuda a evitar o uso excessivo de comentários.

5.8 Inclua protótipos de todas as funções, a fim de tirar proveito dos recursos de verificação do C. Use as diretivas de pré-processador **#include** para obter protótipos de todas as funções da biblioteca padrão a partir dos arquivos de cabeçalho das bibliotecas apropriadas. Use também #**include** para obter os arquivos de cabeçalho que contêm os protótipos de funções usados por você ou pelos membros de sua equipe.

5.9 Algumas vezes, os nomes dos parâmetros são incluídos nos protótipos de funções para fins de documentação. O compilador ignora esses nomes.

5.10 As variáveis usadas apenas em uma determinada função devem ser declaradas como variáveis locais naquela função em vez de variáveis globais.

5.11 Evite nomes de variáveis que desativem (escondam) nomes em escopos externos. Isso pode ser realizado simplesmente evitando o uso de identificadores repetidos em um programa.

Dicas de Portabilidade

5.1 Usar as funções da biblioteca padrão do ANSI C ajuda a tornar os programas mais portáteis.

5.2 Programas que dependem da ordem de cálculo de operandos de operadores diferentes de &&, | |, ?: e o operador vírgula (,) podem funcionar de modo diverso em sistemas com compiladores diferentes.

Dicas de Performance

5.1 O armazenamento automático é um meio de economizar memória porque as variáveis automáticas só existem quando são necessárias. Elas são criadas quando é acionada a função na qual são declaradas e são destruídas quando a função é encerrada.

5.2 O especificador **register** de classe de armazenamento pode ser colocado antes de uma declaração de variável automática para indicar ao compilador que a variável seja conservada em um dos registradores de hardware de alta velocidade do computador. Se as variáveis forem usadas freqüentemente como contadores ou para cálculo de totais, elas podem ser conservadas em registros de hardware, o overhead de carregar repetidamente as variáveis da memória nos registros e armazenar os resultados novamente na memória pode ser eliminado.

5.3 Freqüentemente, as declarações **register** são desnecessárias. Os compiladores otimizados de hoje são capazes de reconhecer as variáveis usadas repetidamente e podem decidir colocá-las em registros sem que haja a necessidade de o programador incluir uma declaração **register.**

5.4 Evite programas recursivos semelhantes ao da função **fibonacci** que resultem em uma "explosão" exponencial de chamadas.

5.5 Evite usar a recursão em situações em que o desempenho é fundamental. As chamadas recursivas são demoradas e consomem memória adicional.

5.6 Um programa com muitas funções — se comparado com um programa monolítico (i.e., constituído por um bloco único) sem funções — executa um número potencialmente grande de chamadas e elas consomem tempo de execução do processador de um computador. Mas os programas monolíticos são difíceis de criar, testar, depurar, conservar e ampliar.

Observações de Engenharia de Software

5.1 Evite reinventar a roda. Quando possível, use as funções da biblioteca padrão do ANSI C em vez de escrever novas funções. Isso reduz o tempo de desenvolvimento de programas.

5.2 Em programas contendo muitas funções, **main (principal)** deve ser implementada como um grupo de chamadas a funções que realizam o núcleo do trabalho do programa.

5.3 Cada função deve se limitar a realizar uma tarefa simples e bem-definida, e o nome da função deve expressar efetivamente aquela tarefa. Isso facilita a abstração e favorece a capacidade de reutilização do software.

5.4 Se você não puder escolher um nome conciso que expresse o que a função faz, é possível que sua função esteja tentando realizar muitas tarefas diferentes. Normalmente é melhor dividir tal função em várias funções menores.

5.5 Uma função não deve ser maior do que uma página. Melhor ainda, uma função não deve ser maior do que metade de uma página. Funções pequenas favorecem a capacidade de reutilização do software.

5.6 Os programas devem ser escritos como conjuntos de pequenas funções. Isso torna os programas mais fáceis de escrever, depurar, manter e modificar.

5.7 Uma função que exige um grande número de parâmetros pode estar realizando tarefas demais. Pense na **1** possibilidade de dividir a função em funções menores que realizem tarefas separadas. O cabeçalho da função deve estar contido em uma linha, se possível.

5.8 O protótipo da função, o arquivo de cabeçalho e as chamadas da função devem concordar quanto ao número, tipo e ordem dos argumentos e parâmetros, e também quanto ao tipo do valor de retorno.

5.9 Um protótipo de função colocado fora de qualquer definição de função se aplica a todas as chamadas daquela função que aparecem depois de seu protótipo no arquivo. Um protótipo de função colocado dentro de uma função se aplica somente às chamadas realizadas naquela função.

5.10 O armazenamento automático é ainda mais um exemplo do princípio do privilégio mínimo. Por que armazenar variáveis na memória e torná-las acessíveis quando na realidade elas não são mais necessárias?

5.11 Declarar uma variável como global em vez de local permite a ocorrência de efeitos colaterais indesejáveis quando uma função modifica acidental ou intencionalmente uma variável, à qual não precisa ter acesso. Em geral, o uso de variáveis globais deve ser evitado, exceto em determinadas situações com exigências especiais de desempenho.

5.12 Qualquer problema que pode ser resolvido recursivamente também pode ser resolvido iterativamente (não-recursivamente). Normalmente, um método recursivo é escolhido em detrimento de um método iterativo quando reflete melhor o problema e resulta em um programa mais fácil de entender e depurar. Outro motivo para escolher um método recursivo é que uma solução iterativa pode não ser aparente.

5.13 Inserir funções de uma forma organizada e hierárquica em programas de computador favorece a boa engenharia de software. Mas isso tem um preço.

Exercícios de Revisão

5.1 Complete as frases seguintes:

a) Um módulo de um programa cm C é chamado ________________.

b) Uma função é chamada com um(a) _____________.

c) Uma variável que é conhecida apenas dentro da função na qual é definida é chamada _-___________________ .

d) A instrução _____________________ em uma função que foi chamada, é utilizada para enviar de volta para a função que chamou, o valor de uma expressão.

e) A palavra-chave __________________ é usada no cabeçalho de uma função para indicar que não é retornado um valor ou para indicar que uma função não contém parâmetros.

f) O (A) __________________ de um identificador é a parte do programa onde o identificador pode ser usado.

g) As três maneiras de retornar o controle de uma função chamada para a chamadora são_________________,_____________e_____________.

h) Um(a) __________________ permite ao compilador verificar o número, os tipos e a ordem dos argumentos passados à função.

i) A função __________________ é usada para produzir números aleatórios.

j) A função __________________ é usada para estabelecer a semente de números aleatórios para randomizar um programa.

k) Os especificadores de classes de armazenamento são ________,________ e _______ .

1) As variáveis declaradas em um bloco ou na lista de parâmetros de uma função são admitidas como pertencendo à classe de armazenamento __________________ , a menos que seja especificado de outra forma.

m) O especificador de classe de armazenamento __________________ é uma recomendação ao compilador para armazenar uma variável em um dos registros do computador.

n) Uma variável declarada externamente a qualquer bloco ou função é uma variável __________________.

o) Para uma variável local em uma função conservar seu valor entre as chamadas da função, ela deve ser declarada com o especificador de classe de armazenamento __________________.

p) Os quatro escopos possíveis para um identificador são _______ , _____________ , ______ e ___________

q) Uma função que chama a si mesma, direta ou indiretamente, é uma função _________ .

r) Geralmente uma função recursiva possui dois componentes: um que fornece um meio de a recursão terminar, examinando um caso __________________, e um que exprime o problema como uma chamada recursiva para um problema ligeiramente mais simples do que o da chamada original.

5.2 Para o programa a seguir, indique o escopo (se é escopo de função, escopo de arquivo, escopo de bloco ouescopo de protótipo de função) de cada um dos seguintes elementos:

a) A variável **x** em **main.**

b) A variável **y** em **cubo.**

c) A função **cubo.**

d) A função **main.**

e) O protótipo da função **cubo.**

f) O identificador **y** no protótipo da função **cubo.**

```
#include <stdio.h>
int cubo(int y) ;
```

```
main() {
int x;
for  (x = 1;  x <= 10; x++)
printf("%d\n", cubo(x));
}
int cubo(int y)
return y * y * y;
}
```

5.3 Escreva um programa que teste se os exemplos de chamadas das funções da biblioteca matemática mostra dos na Fig. 5.2 realmente produzem os resultados indicados.

5.4 Dê o cabeçalho de cada uma das seguintes funções.
a) Função **hipotenusa** que utiliza dois argumentos de ponto flutuante de dupla precisão, **lado1 e lado2,** é retorna um resultado de dupla precisão.
b) Função **menor** que utiliza três inteiros, **x, y, z** e retorna um inteiro.
c) Função **instruções** que não recebe nenhum argumento e não retorna nenhum valor. (Nota: Tais funções são usadas normalmente para exibir instruções para um usuário.)
d) Função **intToFloat** que utiliza um argumento inteiro, **numero,** e retorna um resultado de ponto flutuante.

5.5 Dê o protótipo de cada uma das seguintes funções:
a) A função descrita no Exercício 5.4a.
b) A função descrita no Exercício 5.4b.
c) A função descrita no Exercício 5.4c.
d) A função descrita no Exercício 5.4d.

5.6 Escreva uma declaração para cada um dos seguintes pedidos:

a) Inteiro **contagem** que deve ser mantido em um registro. Inicialize **contagem** com o valor **0 .**
b) Variável de ponto flutuante **ultVal** que deve conservar seu valor entre chamadas da função na qual **é** definida.
c) Inteiro externo **numero** cujo escopo deve estar restrito ao restante do arquivo no qual é definido.

5.7 Encontre o erro em cada um dos seguintes segmentos de programas e explique como ele pode ser corrigido (veja também o Exercício 5.50):
a)
```
int g(void) {
printf("Dentro da funcao g\n");
int h(void) {
printf("Dentro da funcao h\n");
}
}
```
b)
```
int soma(int x,  int y) {
int resultado;
resultado = x + y;
```
c)
```
int soma(int n) {
if  (n == 0) return 0;
else
```

n + soma(n - 1) ;
d) **void f(float a); {**
float a;
printf("%f", a);
}
e) **void produto(void) {**
int a, b, c, result;
printf("Entre com tres inteiros: ") scanf)*"%d%d%d"*, &a, &b, &c); result = a * b * c; printf("O resultado e %d", result); return result;
}

Respostas dos Exercícios de Revisão

5.1 a) Função. b) Chamada de função, c) Variável local, d) **return.** e) **void.** f) Escopo, g) **return;** ou **return expressão;** ou encontrando a chave esquerda de encerramento de uma função. h) Protótipo de função. i) **rand.** j) **srand.** k) **auto, register, extern, static.** 1) Automática, m) **register.** n) Externa, global, o) **static.** p) Escopo de função, escopo de arquivo, escopo de bloco, escopo de protótipo de função, q) recursiva. r) Básico.

5.2 a) Escopo de bloco, b) Escopo de bloco, c) Escopo de arquivo, d) Escopo de arquivo, e) Escopo de arquivo, f) Escopo de protótipo de função.

5.3

```
/* Testando as funções da biblioteca matemática */
#include <stdio.h>
#include <math.h>
main ( ) {
printf("sqrt(%.lf)  = %.1f\n",  900.0,  sqrt(900.0));
printf("sqrt(%.1f)  = %.1f\n",  9.0, sqrt(9.0));
printf("exp(%.1f)  = %f\n",  1.0, exp(1.0));
printf("exp(%.1f)   = %f\n",  2.0, exp(2.0));
printf("log(%f)  =%.1f\n",  2.718282,  log(2.718282));
printf("log(%f)  =%.1f\n",  7.389056,  log(7.389056));
printf("log10(%.1f)  =%.1f\n",  1.0,  log10(1.0));
printf("log10(%.1f)  =%.1f\n",  10.0,  logl10(10.0));
printf("log10(%.1f)  =%.1f\n",  100.0,  log10(100.0));
printf("fabs(%.1f)  =%.1f\n",  13.5,  fabs(13.5));
printf("fabs(%.1f)  =%.1f\n",  0.0, fabs(0.0));
printf("fabs(%.1f)  =%.1f\n",  -13.5,  fabs(-13.5));
printf("ceil(%.1f)  =%.1f\n",  9.2, ceil(9.2));
printf("ceil(%.1f)  =%.1f\n",  -9.8,  ceil(-9.8));
printf("floor(%.1f)   =%.1f\n",  9.2, floor(9.2));
printf("floor(%.1f)  =%.1f\n",  -9.8,  floor(-9.8));
printf("pow(%.1f, %.1f)  = %.1f\n",  2.0,  7.0, pow(2.0, 7.0));
printf("pow(%.1f, %.1f)  = %.1f\n",  9.0,  0.5, pow(9.0, 0.5));
printf ("fmod(%.3f/?&.3f)  = %.3f\n", 13.675,  2.333,  fmod(13.675, 2.333));
printf("sin(%.1f)  = %.1f\n",  0.0, sin(0.0));
printf("cos(%.1f)  = %.1f\n",  0.0, cos(0.0));
printf("tan(%.1f)  =%.1f\n",  0.0, tan(0.0));
}
//Respostas
```

```
sqrt(900.0)  = 30.0
sqrt(9.0)  = 3.0
exp(l.0)  = 2.718282
exp(2.0)  = 7.389056
log(2.718282)   = 1.0
log(7.389056)  = 2.0
logl0(1.0) =0.0
logl0(l00)  = 1.0
logl0(l00.0)  = 2.0
fabs(13.5)  = 13.5
```

```
fabs(0.0) =0.0
fabs(-13.5)  = 13.5
ceil(9.2)  = 10.0
ceil(-9.8) =-9.0
floor(9.2) =9.0
floor(-9.8) =-10.0
pow(2.0,  7.0)  = 128.0
pow(9.0,  0.5)  = 3.0
fmod(13.675/2.333)  = 2.010
sin(0.0)  = 0.0
cos(0.0)  = 1.0
tan(0.0)  = 0.0
```

5.4

a) **double hipotenusa (double lado1, double lado2)**
b) **int menor(int x, int y, int z)**
c) **void instruções(void)**
d) **float intToFloat (int numero)**

5.5

a) **double hipotenusa (double, double);**
b) **int menor(int, int, int);**
c) **void instruções(void)**
d) **float intToFloat(int);**

5.6

a) **register int contagem - 0;**
b) **static float ultVal;**
c) **static int numero;**
Nota: Isso apareceria fora de qualquer definição de função.

5.7

a) Erro: A função **h** é definida na função **g.**
Correção: Mova a definição de **h** para fora da função **g.**
b) Erro: A função deveria retornar um inteiro, mas não faz isso.
Correção: Elimine a variável **resultado** c coloque a seguinte instrução na função: **return x + y;**
c) Erro: O resultado de **n + soma(n - 1)** não é remetido de volta: **soma** retorna um resultado incorreto Correção: Escreva novamente a instrução na cláusula **else** como **return n + soma(n -1);**
d) Erro: O ponto-e-vírgula depois do parêntese direito que encerra a lista de parâmetros e redefinir o parâmetro **a** na definição da função.
Correção: Apague o ponto-e-vírgula depois do parêntese direito da lista de parâmetros e apague a declaração **float a;.**
e) Erro: A função retorna um valor quando não deveria fazer isso. Correção: Elimine a instrução **return.**

Exercícios

5.8 Mostre o valor de x após a execução de cada uma das seguintes instruções:

a) **x = fabs(7.5)**
b) **x = floor(7.5)**
c) **x = fabs(0.0)**
d) **x = ceil(0.0)**
e) **x = fabs (-6.4)**
f) **x = ceil(-6.4)**
g) **x =ceil(-fabs(-8+floor(-5.5)))**

5.9 Uma garagem de estacionamento cobra $2,00 de taxa mínima para estacionar até três horas. A garagem cobra um adicional de $0,50 por hora *ou fração* caso sejam excedidas as três horas. A taxa máxima para qualquer período determinado de 24 horas é $10,00. Admita que nenhum carro fique estacionado por mais de 24 horas. Escreva um programa que calcule e imprima as taxas de estacionamento para três clientes que estacionaram ontem seus carros nessa garagem. Você deve fornecer as horas que cada cliente ficou estacionado. Seu programa deve imprimir os resultados organizados em um formato de tabela e deve calcular e imprimir o total recebido no dia de ontem. O programa deve usar a função **calcula-Taxas** para determinar o valor a ser cobrado de cada cliente. A saída de seu programa deve ter o seguinte formato:

Carro	**Horas**	**Taxa**
1	**1.5**	**2.00**
2	**4.0**	**2.50**
3	**24.0**	**10.00**
TOTAL	**29.5**	**14.50**

5.10 Uma aplicação da função **floor** é arredondar um valor para o inteiro mais próximo.
A instrução **y = floor(x + .5);**
arredondará o número **x** para o inteiro mais próximo e atribuirá o valor a **y.** Escreva um programa que leia vários números e use a instrução anterior para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, imprima o número original e o número arredondado.

5.11 A função **floor** pode ser usada para arredondar um número até uma determinada casa decimal. A instrução
y = floor(x * 10 + .5) / 10;
arredonda ***x*** para décimos (a primeira posição à direita do ponto decimal, ou vírgula). A instrução
y = floor(x * 100 + .5) / 100;
arredonda **x** para centésimos (i.e., a segunda posição à direita do ponto decimal, ou vírgula). Escreva um programa que defina quatro funções para arredondar um número ***x*** de várias maneiras

a) **arredondaParaInteiro(numero)**
b) **arredondaParaDecimos(numero)**

c) **arredondaParaCentesimos(numero)**
d) **arredondaParaMilesimos(numero)**
Para cada valor lido, seu programa deve imprimir o valor original, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

5.12 Responda a cada uma das seguintes perguntas.

a) O que significa escolher números "aleatoriamente"?
b) Por que a função **rand** é útil para simular jogos de azar?
c) Por que randomizar um programa usando **srand?** Sob que circunstâncias não é desejável randomizar?
d) Por que é necessário ajustar e dimensionar a escala dos valores produzidos por **rand?**
e) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?

5.13 Escreva instruções que atribuam inteiros aleatórios à variável *n* nos seguintes intervalos:

a) 1 <= *n* <= 2
b) 1 <= *n* <= 100
c) 0 <= *n* <= 9
d) 1000 <= n <= 1112
e) -1 <= *n* <= 1
f) -3 <= *n* <= 11

5.14 Para cada um dos seguintes conjuntos de inteiros, escreva uma instrução simples que imprima um número aleatório do conjunto.

a) 2, 4, 6, 8, 10.
b) 3,5,7,9, 11.
c) 6, 10, 14, 18,22.

5.15 Defina uma função **hipotenusa** que calcule o comprimento da hipotenusa de um triângulo retângulo, ao serem fornecidos os catetos. Use essa função em um programa para determinar o comprimento da hipotenusa de cada um dos seguintes triângulos. A função deve utilizar dois argumentos do tipo **double** e retornar a hipotenusa com o tipo **double.**

Triângulo	**Lado 1**	**Lado 2**
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

5.16 Escreva uma função **potencialnt (base, expoente)** que retorne o valor de *base^expoente*
Por exemplo, **potencialnt (3,4) =3 * 3 * 3 * 3.** Admita que **expoente** é um inteiro positivo, diferente de zero, e **base** é um inteiro. A função **potencialnt** deve usar **for**

para controlar o cálculo. Não use nenhuma das funções da biblioteca matemática.

5.17 Escreva uma função **múltiplo** que determine, para um par de números inteiros, se o segundo número é múltiplo do primeiro. A função deve ter dois argumentos inteiros e retornar **1** (verdadeiro) se o segundo número for múltiplo do primeiro, e **0** (falso) em caso contrário. Use essa função em um programa que receba uma série de números inteiros.

5.18 Escreva um programa que receba uma série de inteiros e remeta um deles de cada vez para a função **even** que usa o operador resto (modulus) para determinar se um inteiro é par. A função deve utilizar um argumento inteiro e retornar **1** se o inteiro for par e **0** em caso contrário.

5.19 Escreva uma função que mostre, na margem esquerda da tela, um quadrado de asteriscos cujo lado é especificado por um parâmetro inteiro **lado.** Por exemplo, se lado for igual a 4, a função exibe

```
****
****
****
****
```

5.20 Modifique a função criada no Exercício 5.19 para formar um quadrado de qualquer caracter que esteja ***con***tido como parâmetro **fillCharacter.** Dessa forma, se **lado** for igual a **5** e **fillCharacter** for "#". a função deve imprimir

```
#####
#####
#####
#####
#####
```

5.21 Use técnicas similares às descritas nos Exercícios 5.19 e 5.20 para produzir um programa que faça gráficos com uma grande variedade de formas.

5.22 Escreva segmentos de programas que realizem cada uma das tarefas seguintes:
a) Calcule a parte inteira do quociente quando um inteiro **a** é dividido por um inteiro **b.**
b) Calcule o resto inteiro quando um inteiro **a** é dividido por um inteiro **b.**
c) Use os segmentos de programa desenvolvidos em a) e b) para escrever uma função que receba um inteiro entre **1** e **32767** e o imprima com uma série de dígitos, separados por dois espaços. Por exemplo, o inteiro **4562** deve ser impresso como

```
    4  5  6  2
```

5.23 Escreva uma função que obtenha a hora como três argumentos inteiros (para horas, minutos e segundos) e retorne o número de segundos desde a última vez que o relógio "soou as horas". Use essa função para calcular o intervalo de tempo em segundos entre duas horas, ambas dentro de um ciclo de doze horas do relógio.

5.24 Implemente as seguintes funções inteiras:

a) A função **celsius** retorna a temperatura em Celsius equivalente a uma temperatura em Fahrenheit.
b) A função **fahrenheit** retorna a temperatura em Fahrenheit equivalente a uma temperatura em Celsius.
c) Use essas funções para escrever um programa que imprima gráficos mostrando as temperaturas Fahrenheit equivalentes a temperaturas Celsius de 0 a 100 graus e as temperaturas Celsius equivalentes a temperaturas Fahrenheit de 32 a 212 graus. Imprima as saídas organizadas em um formato de tabela que minimize o número de linhas de saída ao mesmo tempo em que permanece legível.

5.25 Escreva uma função que retorne o menor número entre três números de ponto flutuante.

5.26 Diz-se que um número inteiro é um *número perfeito* se a soma de seus fatores, incluindo 1 (mas não o numero em si), resulta no próprio número. Por exemplo, 6 é um número perfeito porque 6 = 1+2 + 3. Escreva uma função **perfeito** que determine se o parâmetro **numero** é um número perfeito. Use essa função em um programa que determine e imprima todos os números perfeitos entre 1 e 1000. Imprima os fatores de cada número encontrado para confirmar que ele é realmente perfeito. Desafie o poder de seu computador testando números muito maiores do que 1000.

5.27 Diz-se que um inteiro é *primo* se for divisível apenas por 1 e por si mesmo. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não são.

a) Escreva uma função que determine se um número é primo.
b) Use essa função em um programa que determine e imprima todos os números primos entre 1 e 10.000. Quantos desses 10.000 números deverão ser realmente testados antes de se ter a certeza de que todos os números primos foram encontrados?
c) Inicialmente você poderia pensar que *n/2 é* o limite superior que deve ser testado para ver se um número é primo, mas você só precisa ir até a raiz quadrada de *n.* Por quê? Escreva novamente o programa e execute-o das duas maneiras. Faça uma estimativa da melhora no desempenho.

5.28 Escreva uma função que utilize um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7631, a função deve retornar 1367.

5.29 O *maior divisor comum (MDC)* de dois inteiros é o maior inteiro que divide precisamente cada um dos dois números. Escreva uma função **mãe** que retorne o maior divisor comum de dois inteiros.

5.30 Escreva uma função **pontosQualif** que receba a média de um aluno e retorne 4 se ela estiver no intervalo 90-100, 3 se a média estiver no intervalo 80-89, 2 se a média estiver no intervalo 70-79, 1 se a média estiver no intervalo 60-69, e 0 se a média for menor do que 60.

5.31 Escreva um programa que simule o lançamento de uma moeda. Para cada lançamento da moeda, o programa deve imprimir **Cara** ou **Coroa.** Deixe o programa lançar a moeda 100 vezes e conte o número de vezes que cada lado da moeda aparece. Imprima os resultados. O programa deve chamar uma função separada **jogada** que não utiliza argumentos e retorna **0** para coroa e **1** para cara. *Nota:* Se o programa simular realisticamente o lançamento da moeda, cada lado da moeda deve aparecer aproximadamente metade do tempo, totalizando cerca de 50 caras e 50 coroas.

5.32 Os computadores estão desempenhando um papel cada vez maior em educação. Escreva um programa que ajudará os alunos da escola do primeiro grau a aprender a multiplicar. Use **rand** para produzir dois inteiros positivos de um dígito. O programa deve então imprimir uma pergunta do tipo

Quanto e 6 vezes 7?

O aluno deve digitar a resposta. Seu programa deve examinar a resposta do aluno. Se ela estiver correta, o programa deve imprimir **"Muito bem!**" e fazer outra pergunta de multiplicação. Se a resposta estiver errada, o programa deve imprimir **"Nao. Tente novamente, por favor**. " e então deixar que o aluno fique tentando acertar a mesma pergunta repetidamente até por fim conseguir.

5.33 O uso dos computadores na educação é conhecido como *instrução assistida por computador (computer-assisted instruction,* CAI). Um problema que surge em ambientes CAI é a fadiga dos alunos. Isso pode ser eliminado variando o diálogo do computador para atrair a atenção do aluno. Modifique o programa do Exercício 5.32 de forma que sejam impressos vários comentários para cada resposta correta e incorreta, como se segue:

Comentários para uma resposta correta

Muito bem!
Excelente!
Bom trabalho!
Certo. Continue assim!

Comentários para uma resposta incorreta

Nao. Tente novamente, por favor. Errado. Tente mais uma vez. Nao desista!
Nao. Continue tentando.

Use o gerador de números aleatórios para escolher um número de 1 a 4 para selecionar o comentário apropriado para cada resposta. Use uma estrutura **switch** com instruções **printf** para imprimir as respostas.

5.34 Sistemas mais sofisticados de instrução assistida por computador controlam o desempenho do aluno durante um período de tempo. Freqüentemente, a decisão de iniciar um novo tópico é baseada no sucesso do aluno no tópico anterior. Modifique o programa do Exercício 5.33 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois de o aluno digitar 10 respostas, seu programa deve calcular a porcentagem de respostas corretas. Se a porcentagem for menor do que 75 por cento, seu programa deve imprimir **"Por favor, solicite**

ajuda extra ao professor" e então ser encerrado.

5.35 Escreva um programa em C que faça o jogo de "adivinhar um número" da forma que se segue: Seu programa escolhe um número para ser adivinhado selecionando um inteiro aleatoriamente no intervalo de 1 a 1000. O programa então escreve:

```
Tenho um numero entre 1 e 1000
Voce pode adivinhar meu numero?
Por favor, digite seu primeiro palpite.
```

O jogador digita então o primeiro palpite. O programa escreve uma das respostas seguintes:

```
1. Excelente! Voce adivinhou o numero! Voce gostaria de tentar
novamente?
2. Muito baixo. Tente novamente.
3. Muito alto. Tente novamente.
```

Se o palpite do jogador estiver incorreto, seu programa deve fazer um loop até que o jogador finalmente acerte o número. Seu programa deve continuar dizendo **Muito alto** ou **Muito baixo** para ajudar o jogador a "chegar" na resposta correta. Nota: A técnica de busca empregada neste problema é chamada *pesquisa binaria.* Falaremos mais sobre isso no próximo problema.

5.36 Modifique o problema do Exercício 5.35 para contar o número de palpites que o jogador fez. Se o número for 10 ou menor, imprima **Ou voce sabe o segredo ou tem muita sorte**! Se o jogador acertar o número em 10 tentativas, imprima **Ahah! Voce sabe o segredo**! Se o jogador fizer mais de 10 tentativas, imprima **Voce deve ser capaz de fazer melhor!** Por que não deve demorar mais do que 10 tentativas? Bem, com cada "bom palpite" o jogador deve ser capaz de eliminar metade dos numeros. Agora mostre por que qualquer número entre 1 e 1000 pode ser adivinhado com 10 palpites ou menos.

5.37 Escreva uma função recursiva **potência (base, expoente)** que, quando chamada, retorna

base*^expoente*

Por exemplo, **potência (3, 4) =3*3*3* 3.** Assuma que **expoente** é um inteiro maior ou igual a 1. *Sugestão:* A etapa de recursão usaria o relacionamento

base*^expoente* **=** ***base* • *base*^(** *expoente-1* **)**

e a condição de encerramento ocorre quando **expoente** for igual a **1** porque *base' =* *base*

5.38 A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13,21,...

começa com os termos 0 e 1 e tem a propriedade de que cada termo subseqüente é a soma dos dois termos precedentes. a) Escreva uma função *não-recursiva* **fibonacci (n)** que calcula o **n**-ésimo número de Fibonacci. b) Determine o maior número de Fibonacci que pode ser impresso em seu sistema. Modifique o

programa da parte a) para usar **double** em vez de **int** para calcular e retornar os números de Fibonacci. Deixe o programa rodar até que seja encerrado por atingir um número excessivamente alto.

5.

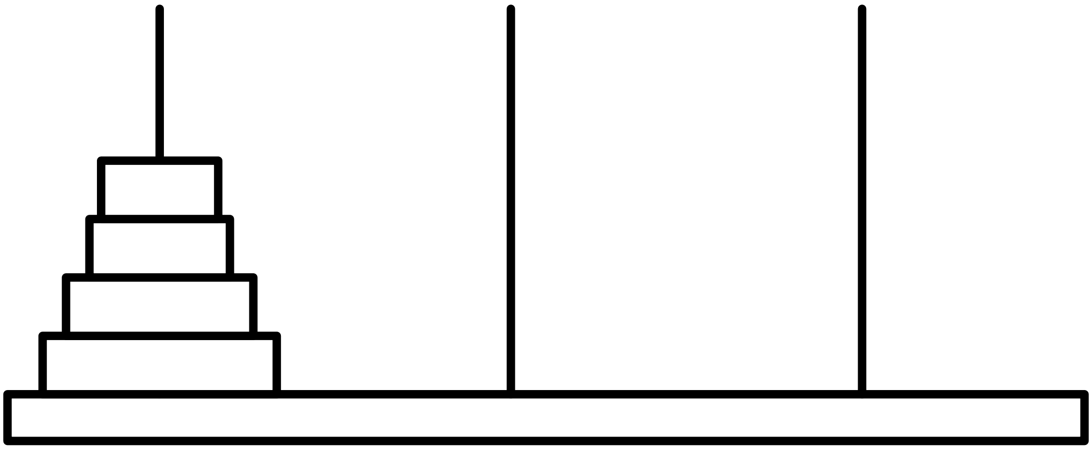


Fig. 5.18 O problema das Torres de Hanói para o caso com quatro discos.

(Torres de Hanói) Todo cientista computacional principiante deve lidar com determinados problemas clássicos, e o problema das Torres de Hanói (veja a Fig. 5.18) é um dos mais famosos. Diz a lenda que em um templo do Extremo Oriente os sacerdotes estão tentando mover uma pilha de discos de um pino para outro. A pilha inicial tinha 64 discos colocados em um pino e organizados na ordem decrescente, da base para o topo. Os sacerdotes estão tentando mover a pilha desse pino para um segundo pino com a restrição de que exatamente um disco deve ser movido de cada vez e em nenhum momento um disco maior pode ser colocado sobre um disco menor. Há um terceiro pino disponível para colocação temporária de discos. Teoricamente o mundo terminará quando os sacerdotes completarem sua tarefa, portanto há pouco estímulo para facilitarmos seus esforços.

Vamos assumir que os sacerdotes estão tentando mover os discos do pino 1 para o pino 3. Desejamos desenvolver um algoritmo que imprimirá a seqüência exata de transferências dc discos de um pino para outro. Se fossemos solucionar este problema com os métodos convencionais, rapidamente nos encontraríamos perdidos lidando com os discos. Em vez disso, se solucionarmos o problema com a recursão em mente, ele se torna imediatamente viável. Mover *n* discos pode ser considerado em termos dc mover apenas *n - I dis*cos (daí a recursão) como se segue:

1   Mova *n* - 1 discos do pino I para o pino 2. usando o pino 3 como área de armazenamento temporário.
2. Mova o último disco (o maior) do pino 1 para o pino 3.
3. Mova os *n* - 1 discos do pino 2 para o pino 3. usando o pino 1 como área dc armazenamento temporário.

O processo termina quando a última tarefa envolver mover *n* = 1 disco, i.e., o caso básico. Isso é realizado movendo simplesmente o disco para o seu destino final, sem a necessidade de uma área de armazenamento temporário.

Escreva um programa para resolver o problema das Torres de Hanói. Use uma função recursiva com quatro parâmetros:

1. O número de discos a serem movidos
2. O pino no qual esses discos estão colocados inicialmente
3. O pino para o qual essa pilha de discos deve ser movida
4. O pino a ser usado como área de armazenamento temporário

Seu programa deve imprimir instruções precisas necessárias para mover os discos do pino inicial para o pino de destino. Por exemplo, para mover uma pilha de três discos do pino 1 para o pino 3, seu programa deve imprimir a seguinte seqüência de movimentos:

1 —> 3 (Isso significa mover o disco do pino 1 para o pino 3.)
1 => 2
3=>2
1 => 3
2 => 1
2=>3
1 -> 3

5.40 Qualquer programa que pode ser implementado recursivamente pode ser implementado iterativamente, embora algumas vezes com dificuldade consideravelmente maior e clareza consideravelmente menor. Tente escrever uma versão iterativa do problema das Torres de Hanói. Se você conseguir, compare sua versão iterativa com a versão recursiva desenvolvida no Exercício 5.39. Investigue questões de desempenho, clareza, e sua habilidade de demonstrar a exatidão do programa.

5.41 (Visualizando a Recursão) É interessante observar a recursão "em ação". Modifique a função fatorial da Fig.5.14 para imprimir sua variável local e parâmetro da chamada recursiva. Para cada chamada recursiva, mostre as saídas em linhas separadas e adicione um nível de recuo. Faça o melhor possível para tornar a saída clara, interessante e significativa. Seu objetivo aqui é desenvolver e implementar um formato de saída que ajude uma pessoa a entender melhor a recursão. Você pode desejar adicionar tais capacidades de exibição a muitos outros exemplos e exercícios de recursão ao longo do texto.

5.42 O maior divisor comum dos inteiros **x** e **y é** o maior inteiro que divide precisamente **x** e **y.** Escreva uma função recursiva **mãe** que retorne o maior divisor comum de **x** e **y.** O maior divisor comum de **x** e **y** é definido recursivamente como se segue: Se **y** for igual a 0, então **mdc (x, y)** é **x;** de outra forma, **mdc (x, y)** é **mdc(y, x%y)** onde *% é* o operador resto (modulus).

5.43 A função **main** pode ser chamada recursivamente? Escreva um programa contendo a função **main.** Inclua a variável **static** local **contagem** inicializada em 1. Pós-incremente e imprima o valor de **contagem** cada vez que **main** for chamada. Rode seu programa. O que acontece?

5.44 Os Exercícios 5.32 a 5.34 desenvolveram um programa de instrução assistida por computador para ensinar multiplicação a um aluno de uma escola do primeiro grau. Este exercício sugere algumas melhorias para esse programa.

a) Modifique o programa para que ele permita ao usuário fornecer um recurso de nível de grau. Um nível de grau 1 significa usar somente números de um único

dígito nos problemas, um nível de grau dois significa usar números maiores de dois dígitos etc.
b) Modifique o programa para que ele permita ao usuário selecionar o tipo de problemas aritméticos que deseja estudar. Uma opção igual a 1 significa apenas problemas de adição, 2 significa apenas problemas
de subtração, 3 significa apenas problemas de multiplicação, 4 significa problemas apenas de divisão e 5 significa misturar aleatoriamente os problemas de todos esses tipos.

5.45 Escreva uma função **distancia** que calcule a distância entre dois pontos (x1, y1) e (x2, y2). Todos os números e valores de retorno devem ser do tipo **float.**

5.46 O que faz o seguinte programa?

```
main() {
int c ;
if  ((c = getchar( ))   != EOF) { main();
print("%c", c);
}
return 0;
```

5.47 O que faz o seguinte programa? **int mistério (int, int);**

```
main(  ) {
int x, y;
printf("Entre com dois inteiros:"); scanf("%d%d",  &x, &y);
printf("O resultado e %d\n", mistério(x, y)); return 0;
}
/*0 parâmetro b deve ser um inteiro positivo
para evitar recursão infinita */ int mistério(int a,  int b) {
if (b == 1) return a; else
return a + mistério(a, b - 1);
}
```

5.48 Depois de determinar o que faz o programa do Exercício 5.47, modifique-o para funcionar adequadamente após remover a restrição de o segundo argumento ser não-negativo.

5.49 Escreva um programa que teste o máximo possível das funções da biblioteca matemática na Fig. 5.2. Experimente cada uma das funções fazendo com que seu programa imprima tabelas com os valores de retorno para vários valores de argumentos.

5.50 Encontre o erro em cada um dos seguintes segmentos de programas e explique como corrigi-los:

a) **float cubo (float); /* protótipo da funcao */**
cubo (float numero) /* definição da funcao */ {
return numero * numero * numero;
b) **register auto int x = 7;**
c) **int randomNumber = srand();**
d) **float y = 123.45678; int x;**
x = y;

printf("%f\n",   (float) x);
e) **double quadrado (double numero) {**
double numero;
return numero * numero;
}
f) **int soma (int n) {**
if (n == 0)
return 0;
else
return n + soma(n);
}

5.51 Modifique o programa do jogo de craps da Fig. 5.10 para permitir apostas. Transforme em uma função à parte do programa que executa um jogo de craps. Inicialize a variável **bankBalance** com 1000 dólares. Peça ao jogador para entrar com uma aposta **(wager).** Use um loop **while** para verificar se **wager** é menor ou igual a **bankBalance** e, se não for, peça ao jogador para fornecer um novo valor de aposta até que um valor válido para **wager** seja fornecido. Depois de ser informado um valor correto para **wager,** execute um jogo de craps. Se o jogador vencer, aumente **bankBalance** com o valor de **wager** e imprima o novo **bankBalance.** Se o jogador perder, subtraia o valor de **wager** do total em **bankBalance,** imprima o novo **bankBalance,** verifique se **bankBalance** se tornou zero e, em caso positivo, imprima a mensagem **"Sinto muito. Voce esta arruinado!** ". Com o passar do jogo, imprima várias mensagens para criar um "interlocutor" como **"Oh, voce vai perder tudo, hem?"** ou **"Isso mesmo, teste sua sorte**!" ou **"Voce e o máximo. Chegou a hora de lucrar!".**

6

Arrays

Objetivos

- Apresentar a estrutura de dados do array.
- Entender o uso de arrays para armazenar, classificar e pesquisar listas e tabelas de valores.
- Entender como declarar um array, inicializar um array e fazer referência aos elementos de um array.
- Ser capaz de passar arrays a funções.
- Entender as técnicas básicas de classificação.
- Ser capaz de declarar e manipular arrays com vários subscritos.

Com soluços e lágrimas ele separou Os de tamanho maior...
Lewis Carroll

Tente terminar e nunca vacile;
Nada é mais difícil, mas a pesquisa descobrirá.
Robert Herrick

Vai pois agora, escreve isto numa tábua perante eles, e aponta-o num livro.
Isaías 30:8

Está preso em minha memória,
E você mesmo deve guardar a chave.
William Shakespeare

Sumário

6.1 Introdução

Este capítulo serve como uma introdução para o importante tópico de estruturas de dados. Os *arrays* são estruturas de dados que consistem em itens de dados do mesmo tipo, relacionados entre si. No Capítulo 10, analisamos a noção de **struct** (estrutura) do C — uma estrutura de dados que consiste em itens de dados relacionados entre si, provavelmente de tipos diferentes. Os arrays e as estruturas são entidades "estáticas", já que permanecem do mesmo tamanho ao longo de toda a execução do programa (eles podem, obviamente, ser da classe de armazenamento automático e assim criados e destruídos cada vez que o programa entra e sai dos blocos nos quais são definidos). No Capítulo 12, apresentamos estruturas dinâmicas de dados, como as listas, filas, pilhas e árvores que podem crescer ou reduzir de tamanho à medida que o programa é executado.

6.2 Arrays

Um array é um grupo de locais da memória relacionados pelo fato de que todos têm o mesmo nome e o mesmo tipo. Para fazer referência a um determinado local ou elemento no array, especificamos o nome do array e o *número da posição* daquele elemento no array.

A Fig. 6.1 mostra um array inteiro chamado c. Esse array contém doze *elementos.* Pode-se fazer referência a qualquer um desses elementos fornecendo o nome do array seguido do número da posição do elemento desejado entre colchetes ([ ]). O primeiro elemento em qualquer array é o *elemento zero.* Dessa forma, o primeiro elemento do array c é chamado c [0], o segundo elemento do array c é chamado c [1], o sétimo elemento do array c é chamado c [6] e, em geral, o elemento *i* do array c é chamado c [i-1]. Os nomes dos arrays seguem as mesmas convenções para os nomes de outras variáveis.

O número de posição contido entre colchetes é chamado mais formalmente um *subscrito.* Um subscrito deve ser um inteiro ou uma expressão inteira. Se um programa usar uma expressão inteira como subscrito, a expressão é calculada para determinar o valor do subscrito. Por exemplo, se a = 5 e b = *6.* a instrução

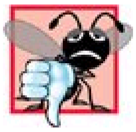
c[a + b] += 2;

adiciona 2 ao elemento de array c [11]. Observe que o nome de um array com subscritos é um 1 value — ou seja, pode ser usado no lado esquerdo de uma atribuição.

Vamos examinar mais detalhadamente o array c na Fig. 6.1.0 *nome* do array é c. Seus doze elementos são denominados c[0] ,c[l] ,c [2], ... ,c [11]. O *valor de* c [0] é -45, o valor de c[1] é 6, o valor de c [2] é 0, o valor de c [7] é 62 e o valor de c [11] é 7 8. Para imprimir a soma dos valores contidos nos três primeiros elementos do array c, escreveríamos

printf("%d", c[0]   + c[1] + c[2]);

Para dividir por 2 o valor do sétimo elemento do array c e atribuir o resultado à variável x, escreveríamos x = c[6]/2;



Erro comun de programação 6.1

É importante observar a diferença entre o "sétimo elemento do array" e o "elemento número sete do array". Como os subscritos dos arrays iniciam em 0, o "sétimo elemento do array" tem subscrito 6, ao passo que o "elemento número sete do array" tem subscrito 7 e é, na realidade, o oitavo elemento do array. Esse fato é uma fonte de erros de "diferença-um".

Na realidade, os colchetes usados para conter o subscrito de um array são considerados um operador pela linguagem C. Eles possuem o mesmo nível de precedência que os parênteses. A tabela da Fig. 6.2 mostra a precedência e a associatividade dos operadores apresentados até agora no texto. Eles são mos trados de cima para baixo na ordem decrescente de precedência.

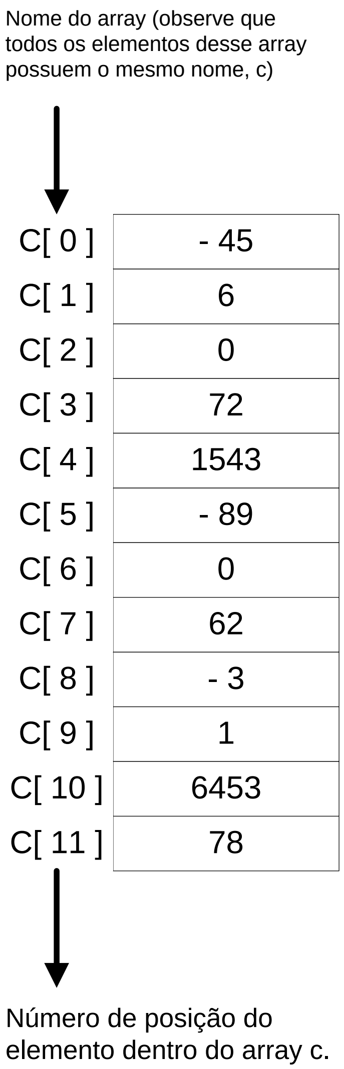


Fig. 6.1 Um array com doze elementos.

6.3 Declarando Arrays

Os arrays ocupam espaço na memória. O programador especifica o tipo de cada elemento e o numero de elementos exigidos por array de forma que o computador possa reservar a quantidade apropriada de memória. Para dizer ao computador para reservar 12 elementos para o array inteiro **c,** é usada a declaração

int c[12];

Pode-se reservar memória para vários arrays com uma única declaração. Para reservar 100 elementos para o array **b** e 27 elementos para o array **x,** usa-se a seguinte declaração:

int b[100], x[27];

Os arrays podem ser declarados para conter outros tipos de dados. Por exemplo, pode ser usado um array do **tipo char** para armazenar uma string de caracteres. As strings de caracteres e seu relacionamento com os arrays **são** analisados no Capítulo 8. O relacionamento entre ponteiros e arrays é analisado no Capítulo 7.

Operadores	Associatividade	Tipo
( ) [ ]	Da esquerda para direita	maior
++ -- ! (*tipo*)	Da direita para esquerda	Unário
* / %	Da esquerda para direita	Multiplicativo
+ -	Da esquerda para direita	Aditivo
< <= > >=	Da esquerda para direita	Relacional
== !=	Da esquerda para direita	Igualdade
&&	Da esquerda para direita	E lógico
			Da esquerda para direita	Ou lógico
?:	Da direita para esquerda	Condicional
= += -= *= /= %=	Da direita para esquerda	Atribuição
,	Da esquerda para direita	Vírgula

Fig. 6.2 Precedência dos operadores,

6.4 Exemplos usando Arrays

O programa da Fig. 6.3 usa uma estrutura de repetição **for** para inicializar com zeros os elementos de um array inteiro **n** de dez elementos, e imprime o array sob a forma de uma tabela.

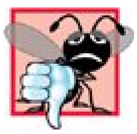
Observe que decidimos não colocar uma linha em branco entre a primeira instrução **printf** e a estrutura **for** na Fig. 6.3 porque elas estão intimamente relacionadas. Nesse caso, a instrução **printf** exibe os cabeçalhos das colunas para as duas colunas impressas na estrutura **for.** Freqüentemente, os programadores omitem a linha em branco entre uma estrutura **for** e uma instrução **printf** a ela relacionada.

Os elementos de um array também podem ser inicializados na declaração do array com um sinal de igual e uma lista de *inicializadores* separada por vírgulas (entre chaves). O programa da Fig. 6.4 inicializa *um* array inteiro com dez valores e o imprime sob a forma de uma tabela.

Se houver menos inicializadores do que o número de elementos do array, os elementos restantes são inicializados automaticamente com o valor zero. Por exemplo, os elementos do array **n** da Fig. 6.3 poderiam ser inicializados com o valor zero por meio da declaração

int n[10]   = {0};

que inicializa explicitamente o primeiro elemento com zero e automaticamente inicializa os nove elementos restantes também com zero porque há menos inicializadores do que elementos do array. É importante lembrar que os arrays não são inicializados automaticamente com zero. O programador deve inicializar pelo menos o primeiro elemento com zero para que os elementos restantes sejam automaticamente zerados. Esse método de inicializar elementos de array com **0** é realizado em tempo de compilacão .O método usado na Fig. 6.3 pode ser feito repetidamente durante a execução do programa.



Erro comun de programação 6.2

Esquecer de inicializar os elementos de um array que precisam ser inicializados.

```
/* inicializando um array */
#include <stdio.h>

main() {
        int n[10],  i;
        for (i = 0; i <= 9; i++) /* inicializa o array */
        n[i]   = 0;
        printf("%s%13s\n",   "Elemento", "Valor"); for(i =0;i<=9;i++) /*
imprime o array */
        printf ("%7d%13d\n",  i, n[i]);
        return 0;
}
```

Elemento	Valor
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Fig. 6.3 Inicializando com zero os elementos de um array.

```
/* Inicializando um array com uma declaração */
#include <stdio.h>
main()
{
        int i,  n[10]  = {32,  27,  64,  18,  95,  14,  90,  70,  60, 37};
        printf("%s%13s\n",   "Elemento", "Valor");
        for(i = 0;  i <= 9; i++)
        printf("%7d%13d\n",  i, n[i]);
        return 0;
}
```

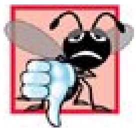
Elemento	Valor
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Fig. 6.4 Inicializando os elementos de um array com uma declaração.

A seguinte declaração de array

int n[5] = {32 ,27 ,67 ,18 ,95 ,14};

causaria um erro de sintaxe porque há seis inicializadores e apenas 5 elementos no Array.



Erro comun de programação 6.3

Fornecer mais inicializadores para um array do que o numero de seus elementos é um erro de sintaxe.

Se o tamanho do arrau for omitido de uma declaração com uma lista de inicializadores, o número de elementos do array será o número de elementos da lista de inicializadores. Por exemplo,

int n[] = {1 , 2 , 3 , 4 , 5};

criaria um array com cinco elementos.

O programa da Fig. 6.5 inicializa os dez elementos de um array **s** com os valores **2, 4, 6, ..., 20** e imprimi o array em forma de tabela. Os valores são gerados multiplicando o cantador de loops por 2 e somando 2 ao produto anterior.

A diretiva **#define** do pré-processador é apresentada nesse programa. A linha

#define TAMANHO 10

Define uma *constante simbólica* **TAMANHO** cujo valor é 10. Uma constante simbólica é um identificador que é substituído por um *texto de substituição* pelo pré-processador da linguagem C antes do pro grama ser compilado. Quando o programa é pré-processado, todas as ocorrências da constante simbólica **TAMANHO** são substituídas pelo texto de substituição **10.** Usar constantes simbólicas para especificar tamanhos de arrays torna os programas mais *flexíveis (escaláveis).* Na Fig. 6.5, o primeiro loop **foi** preencheria um array de 1000 elementos simplesmente modificando de **10** para **1000** o valor de **TAMANHO** na diretiva **#define.** Se a constante simbólica **TAMANHO** não tivesse sido utilizada, teríamos de modificar o programa em três locais diferentes para adaptá-lo ao array de 1000 elementos. À medida que os programas se tornam maiores, essa técnica se torna mais útil para escrever programas claros.

```
/* Inicializa os elementos do array com os inteiros pares de 2 a 20 */
#include <stdio.h>
#define TAMANHO 10
{
        int s[TAMANHO], j;

        for(j = 0;  j <= 9; j++) /* Define os valores */
                s[j] = 2 + 2 * j;

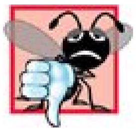
        printf("%s%13s\n","Elemento", "Valor");

        for(j = 0;  j <= 9; j++) /* Imprime os valores */
                printf("%8d%13s\n", j, s[j]);

        return 0;
}
```

Elemento	Valor
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

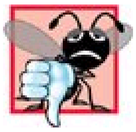
Fig. 6.5 Gerando os valores a serem colocados nos elementos de um array,



Erro comun de programação 6.4

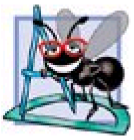
*Encerrar uma declaração **#define** ou **#include** com ponto-e-vírgula. Lembre-se de que as diretivas do pré-processador não são instruções da linguagem C.*

Se a diretiva **#define** anterior fosse encerrada com um ponto-e-vírgula, todas as ocorrências da constante simbólica **TAMANHO** no programa seriam substituídas pelo texto **10**; pelo pré-processador Isso poderia levar a erros de sintaxe em tempo de compilação, ou a erros lógicos em tempo de execução Lembre-se de que o pré-processador não é C — ele é apenas um manipulador de textos.



Erro comun de programação 6.5

Atribuir um valor a uma constante simbólica em uma instrução executável é um erro de sintaxe. Uma constante simbólica não é uma variável. O compilador não reserva espaço para ela, como faz com as variáveis que possuem valores em tempo de execução.



Observação de engenharia de software 6.1

Definir o tamanho de cada array com uma constante simbólica torna os programas mais flexíveis.



Boa prática de programação 6.1

Use apenas letras maiúsculas para nomes de constantes simbólicas. Isso faz essas constantes se destacarem no programa e lembra o programador que as constantes simbólicas não são variáveis.

O programa da Fig. 6.6 soma os valores contidos em um array **a,** que possui doze elementos inteiros A instrução no corpo do loop **for** faz a totalização.

```
/* Calcula a soma dos elementos de um array */
#include <stdio.h>
#define TAMANHO 12
main() {
        int a[TAMANHO] = {1,3,5,4,7,2,99,16,45,67,89,45}, i, total =0;

        for  (i = 0;   i <= TAMANHO - 1; i++)
                total += a[i];

        printf("A soma dos elementos do array e %d\n", total);
        return 0;
}
```

```
A soma dos elementos do array e 383
```

Fig. 6.6 Calculando a soma dos elementos de um array,

Nosso próximo exemplo usa arrays para resumir os resultados dos dados coletados em uma pesquisa. Considere o seguinte enunciado de problema.

Foi perguntado a quarenta alunos o nível de qualidade da comida na cantina estudantil, em uma escala de 0 a 10 (1 significa horrorosa e 10 significa excelente). Coloque as quarenta respostas em um array inteiro e resu-ma os resultados da pesquisa.

Essa é uma aplicação típica de arrays (veja a Fig. 6.7). Queremos resumir o número de respostas de cada tipo (i.e., 1 a 10). O array **respostas** possui 40 elementos de respostas dos alunos. Usamos um array **frequencia,** com onze elementos, para contar o número de ocorrências de cada resposta. Ignoramos o primeiro elemento, **freqüência [0]**, porque é mais lógico fazer com que a resposta 1 incremente **freqüência [1]** do que **freqüência [0]**. Isso permite usar diretamente cada resposta como subscrito do array **frequencia.**



Boa prática de programação 6.2

Procure obter clareza nos programas. Algumas vezes vale a pena prejudicar o uso mais eficiente da memória em prol de tornar os programas mais claros.

```
/* Programa de votação estudantil */
#include <stdio.h>
#define TAMANHO_RESPOSTAS 40
#define TAMANHO_FREQUENCIA 11

main()
{
        int opinião, nivel;
        int respostas[TAMANHO_RESPOSTAS]  =
        {1,2,6,4,8,5,9,7,8,10,1,6,3,8,6,10,3,8,2,7,6,5
        ,7,6,5,7,6,8,6,7,5,6,6,5,6,7,5,6,4,8,6,8,10};
        int freqüência[TAMANHO_FREQUENCIA]  = {0};

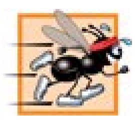
        for  (opiniao = 0; opiniao <= TAMANHO_RESPOSTAS - 1; opiniao++)
                ++frequencia[respostas[opiniao]];

        printf("%s%17s\n",   "Nivel", "Frequencia");
        for (nivel = 1; nivel <= TAMANHO_FREQUENCIA - 1; nivel++)
                printf("%5d%17d\n", nivel,  frequencia[nivel]);

        return 0;
}
```

Nivel	**Frequencia**
1	**2**
2	**2**
3	**2**
4	**2**
5	**5**
6	**11**
7	**5**
8	**7**
9	**1**
10	**3**

Fig. 6.7 Um programa simples de análise de votação estudantil,



Dica de desempenho 6.1

Algumas vezes as considerações de desempenho são muito mais importantes do que as considerações de clareza.

O primeiro loop **for** obtém uma resposta de cada vez do array **respostas** e incrementa um dos dez contadores (**frequencia[1]** a **frequencia[10]**) no array frequencia. A instrução prin-cipal do loop é

++frequencia[respostas[opiniao]];

Essa instrução incrementa o contador **frequencia** apropriado dependendo do valor **respostas[opiniao]**. Por exemplo, quando a variável do contador **opiniao** for 0, **respostas[opiniao]** é 1, portanto **++frequencia [respostas [opiniao]]** ; é na verdade interpretada como

++frequencia[1];

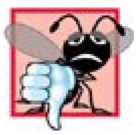
que incrementa o elemento um do array. Quando **opiniao** for **1, respostas [opiniao]** será **2**, portanto **++frequencia [respostas [opiniao]]** ; é interpretado como

++frequencia[2] ;

que incrementa o elemento dois do array. Quando **opiniao** for **2**, **respostas[opiniao]** será **6**, portanto **++frequencia [respostas [opiniao]] ;** é interpretado como

++frequencia[6];

que incrementa o elemento seis do array, e assim por diante. Observe que, independentemente do número de respostas processadas, apenas um array de onze elementos é necessário (ignorando o elemento zero) para resumir os resultados. Se os dados tivessem valores inválidos como 13, o programa tentaria adicionar **1** a **frequencia[13]**. Isso estaria além dos limites do array. *A linguagem C não verifica os limites do array para evitar que o computador faça referência a um elemento não-existente.* Assim um programa pode ultrapassar o final de um array sem emitir qualquer aviso. O programador deve certeza de que todas as referências ao array permanecem dentro de seus limites.



Erro comun de programação 6.6

Fazer referência a um elemento além dos limites de um array.



Boa prática de programação 6.3

Ao fazer um loop em um array, os subscritos nunca devem ser menores do que 0 e sempre devem ser menores do que o número total de elementos do array (tamanho - 1). Certifique-se de que a condição de término do loop evita o acesso a elementos fora desses limites.



Boa prática de programação 6.4

Mencione o maior subscrito de um array em uma estrutura ***for*** *para ajudar a eliminar erros de "diferença-um".*



Boa prática de programação 6.5

Os programas devem verificar a exatidão de todos os valores de entrada para evitar que informações inadequadas afetem os cálculos ali realizados.

Dica de desempenho 6.2

Os efeitos (geralmente graves) da referência a elementos situados fora dos limites de um array dependem dos sistemas.

Nosso próximo exemplo (Fig. 6.8) lê os números de um array e faz um gráfico das informações na forma de um gráfico de barras ou histograma — cada número é impresso e então uma barra constituída

```
/* Programa para imprimir histograma */
#include <stdio.h>
#define TAMANHO 10
main() {
        int n[TAMANHO] = {19,3,15,7,11,9,13,5,17,1};
        int i, j;

        printf("%s%13s%"l7s\n",   "Elemento",   "Valor", "Histograma");

        for  (i = 0;   i <= TAMANHO -  1;  i++) {
                printf("%8d%13d ",  i, n[i]);

                for  (j  = 1;  j <= n[i];  j++)    /* imprime uma barra */
                        printf("%c",   ' * ');

                printf("\n");
        }
        return 0;
}
```

```
Elemento      Valor         Histograma
   0           19       *******************
   1            3       ***
   2           15       ***************
   3            7       *******
   4           11       ***********
   5            9       *********
   6           13       *************
   7            5       *****
   8           17       *****************
   9            1       *
```

Fig. 6.8 Um programa que imprime histogramas.

No Capítulo 5 afirmamos que mostraríamos um método mais elegante de escrever o programa do jogo de dados da Fig. 5.10. O problema era rolar um dado de seis faces 6000 vezes para verificar se o criador de números aleatórios produzia realmente números aleatórios. Uma versão desse programa utilizando arrays é mostrada na Fig. 6.9.

Até este momento analisamos apenas arrays inteiros. Entretanto, os arrays são capazes de conter dados de qualquer tipo. Analisamos agora o armazenamento de strings em arrays de caracteres. Até agora, a única capacidade de processamento de strings que utilizamos foi imprimir uma string com **printf.** Uma string como **"hello**" é, na realidade, um array de caracteres isolados em C.

Os arrays de caracteres possuem várias características exclusivas. Um array de caracteres pode ser inicializado usando uma string literal. Por exemplo, a declaração

char string1[] = "primeiro"

inicializa os elementos do array **string1** com os valores de cada um dos caracteres da string literal **"primeiro**". O tamanho do array **string1** na declaração anterior é determinado pelo compilador, com base no comprimento da string. É importante notar que a string " **primeiro**" contém oito caracteres *mais* um caractere especial de término de string, chamado *caractere nulo.* Dessa forma, o array **string1** contem na verdade seis elementos. A representação da constante do caractere nulo é ' \ **0** '. Todas as strings em C terminam com esse caractere. Um array de caracteres representado por uma string deve sempre ser declarado com tamanho suficiente para conter o número de caracteres da string mais o caractere nulo de terminação.

```
/* Rolar um dado de seis faces 6000 vezes */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAMANHO 7
main() {
        int face,  jogada,  frequencia[TAMANHO]  = {0};

        srand(time(NULL));

        for (jogada = 1; jogada <= 600 0; jogada++) {
                face = rand() % 6 + 1 ;
                ++frequencia[face]; /* substitui o switch de 20 linhas"
        }                          /* da Fig. 5.8 */~

        printf("%s%18s\n",   "Face", "Frequencia");

        for  (face = 1;  face <= TAMANHO - 1; face++)
                printf("%4d%18d\n",  face,  frequencia[face]);
        return 0;
}
```

Face	**Frequencia**
1	**1013**
2	**1028**
3	**952**
4	**983**
5	**987**
6	**1037**

Fig. 6.9 Programa de jogo de dados usando arrays em vez de switch.

Os arrays de caracteres também podem ser inicializados com constantes de caracteres isolados em uma lista de inicializadores. A declaração anterior é equivalente

char stringl[] = {'p', 'r', 'i', 'm', 'e', 'i', 'r', 'o', '\0'};

Em conseqüência de as strings serem na realidade arrays de caracteres, podemos ter acesso direto a caracteres isolados de uma string usando a notação de subscritos de arrays. Por exemplo, **string1** [0] é o caractere **'p'** e **stringl** [3] é o caractere **'m'.**

Podemos também fornecer uma string diretamente a um array de caracteres a partir do teclado usando **scanf** e a especificação de conversão ***%s.*** Por exemplo, a declaração

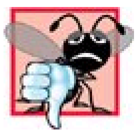
char string2[20];

cria um array de caracteres capaz de armazenar uma string de 19 caracteres e um caractere nulo de tér-mino. A instrução

scanf(*"%s"*, string2);

lê uma string do teclado e a armazena em **string2.** Observe que o nome do array é passado para **scanf** sem ser precedido por &, usado com outras variáveis. O & é usado normalmente para fornecer a **scanf** o local de uma variável na memória de forma que o valor possa ser armazenado ali. Na Seção 6.5, analisamos como passar arrays a funções. Veremos que o nome de um array é o endereço do início do array; portanto, o & não é necessário.

É responsabilidade do programador assegurar que o array para o qual a string é lida pode conter qualquer string que o usuário digite no teclado. A função **scanf** lê caracteres do teclado até que o primeiro caractere em branco seja encontrado — para ela o tamanho do array não tem importância. Dessa forma, **scanf** poderia ir além do final do array.



Erro comun de programação 6.7

*Não fornecer a **scanf** um array de caracteres com comprimento suficiente para armazenar uma string digitada no teclado pode resultar em perda de dados de um programa e outros erros em tempo de execução.*

Um array de caracteres representando uma string pode ser impresso com **printf** e o especificador **de** conversão ***%s.*** O array **string2** é impresso com a instrução

printf("%s\n", string2);

observe que para **printf,** assim como para **scanf,** não importa o tamanho do array de caracteres. Os caracteres da string são impressos até que seja encontrado um caractere nulo de terminação.

A Fig. 6.10 demonstra como inicializar um array de caracteres com uma string literal, ler uma string e armazená-la em um array de caracteres, imprimir um array de caracteres como string e ter acesso a caracteres individuais de uma string.

```
/* Tratando arrays de caracteres como strings */
#include <stdio.h>
main() {
        char string[20],  string2[]  = "string literal";
        int i;

        printf("Entre com uma string: ");
        scanf("%s", stringl);
        printf("string1 e: %s\nstring2 e: %s\n"
            "string1 com espaços entre caracteres e:\n", stringl,  strlng2);

        for (i = 0;  stringl[i]   !=  '\0'; i++)
                printf("%c",  stringl[i]);

        printf("\n");

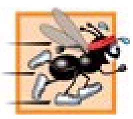
        return 0;
}
```

```
Entre com uma string: Hello there
stringl e: Hello
string2 e:  string literal
stringl com espaços entre caracteres e:
Hello
```

Fig. 6.10 Tratando arrays de caracteres como strings,

A Fig. 6.10 usa uma estrutura **for** para fazer um loop através do array **string1** e imprimir os caracteres isolados, separados por espaços, usando a especificação de conversão ***%c.*** A condição na estrutura **for, string1 [i] ! = '\0',** será verdadeira enquanto o caractere nulo de terminação não for encontrado na string.

O Capítulo 5 analisou o especificador de classe de armazenamento **static.** Uma variável local **static** em uma definição de função existe durante a execução do programa, mas só é visível no corpo da função. Podemos aplicar **static** à declaração local do array para que este não seja criado e inicializado sempre que a função for chamada e não seja destruído sempre que a função acabar de ser executada pelo programa. Isso reduz o tempo de execução do programa, particularmente para programas com funções que são chamadas freqüentemente e contêm arrays grandes.

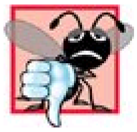


Dica de desempenho 6.3

*Em funções que contêm arrays automáticos **(automatic)** onde se entra e sai do escopo daquela função freqüentemente, faça com que o array seja **static,** para que ele não seja criado sempre que a função for chamada.*

Os arrays declarados estáticos **(static)** são inicializados automaticamente uma única vez no tempo de compilação. Se um array **static** não for inicializado explicitamente pelo programador, será inicializado com zeros pelo compilador.

A Fig. 6.11 demonstra a função **staticArraylnit** com um array local declarado **static** e a função **automaticArraylnit** com um array local automático. A função **staticArraylnit** é chamada duas vezes. O array local **static** na função é inicializado com zeros pelo compilador. A função imprime o array, adiciona 5 a cada elemento e imprime o array novamente. Na segunda vez em que a função é chamada, o array **static** contém os valores armazenados durante a primeira execução da função. A função **automaticArraylnit** também é chamada duas vezes. Os elementos do array local automático na função são inicializados com os valores 1, 2 e 3. A função imprime o array, adiciona 5 a cada elemento e imprime o array novamente. Na segunda vez em que a função é chamada, os elementos do array são inicializados com **1,** 2 e 3 novamente porque o array tem tempo automático de armazenamento.



Erro comun de programação 6.8

*Supor que os elementos de um array local declarado **static** são inicializados com zeros sempre que a função na qual o array é declarado for chamada.*

6.5 Passando Arrays a Funções

Para passar um argumento array a uma função, especifique o nome do array sem colocar colchetes. Por exemplo, se o array **temperaturaPorHora** for declarado como

int temperaturaPorHora[24];

a instrução de chamada da função

modificaArray(temperaturaPorHora, 24);

passa o array **temperaturaPorHora** e seu tamanho para a função **modificaArray.** Freqüentemente, ao passar um array para uma função, o tamanho do array é passado para que ela possa processar o número total de elementos do array.

A linguagem C passa automaticamente os arrays às funções usando chamadas por referência simuladas — as funções chamadas podem modificar os valores dos elementos nos arrays originais dos locais que fazem as chamadas. Na verdade, o nome do array é o endereço do primeiro elemento do array! Por ser passado o endereço inicial do array, a função chamada sabe precisamente onde o array está armazenado. Portanto, quando a função chamada modifica os elementos do array em seu corpo de função, os elementos reais estão sendo modificados em suas posições originais da memória.

```
/* Os arrays static sao inicializados com zeros */
#include <stdio.h>

void staticArraylnit(void);
void automaticArraylnit(void);

main() {
        printf("Primeira chamada de cada função:\n");
        staticArraylnit();
        automaticArraylnit();

        printf("\n\nSegunda chamada de cada função:\n");
        staticArraylnit();
        automaticArraylnit();

        return 0;
}

/* função para demonstrar um array local static */
void staticArraylnit(void)
{
    static int a[3];
    int i;

    printf("\nValores de staticArraylnit ao entrar:\n");

    for  (i = 0;   i <= 2; i++)
```

```
                printf("array1[%d]  = %d ",  i, a[i]);

        printf("\nValores de staticArraylnit ao sair:\n");

        for  (i = 0;  i <= 2; i++)
                printf("arrayl[%d]  = %d *,  i,  a[i]   += 5);
}

/* função para demonstrar um array local automatic */
void automaticArraylnit(void)
{
        int a[3]  = {1,  2, 3};
        int i ;

        printf("\n\nValores de automaticArraylnit ao entrar:\n");

        for  (i = 0;  i <= 2; i++)
                printf("arrayl[%d]   = %d ",  i, a[i]);

        printf("\n\nValoresde automaticArraylnit ao sair:\n");

        for  (i = 0;   i <= 2; i++)
                printf("arrayl[%d]  = %d ",  i,  a[i]  += 5);
}
```

```
Primeira chamada de cada função:
Valores de staticArraylnit ao entrar:
array1[0]  = 0       array1[1]  = 0       array1[2] = 0
Valores de staticArraylnit ao sair:
array1[0]  = 5       array1[1]  = 5       array1[2]  = 5
Valores de automaticArraylnit ao entrar:
 array1[0]  = 1       array1[1]  = 2       array1[2]  = 3
Valores de automaticArraylnit ao sair:
array1[0]  = 6         arraylt[1]  = 7      arrayl[2]  = 8

Segunda chamada de cada função:
Valores de staticArraylnit ao entrar:
array1[0] = 5       array1[1]  = 5       arrayl[2]  = 5
Valores de staticArraylnit ao sair:
array1[0]  = 10       array1[1]  = 10       arrayl[2]  = 10
Valores de automaticArraylnit ao entrar:
 array1[0]  = 1        array1[1]  = 2 arrayl[2]  = 3
Valores de automaticArraylnit ao sair:
array1[0] = 6       array1[1]  = 7       arrayl[2]  = 8
```

Fig. 6.11 Os arrays estáticos **(static)** são inicializados automaticamente com zeros se não forem inicializados explicitamente pelo programador.

A Fig. 6.12 demonstra que o nome de um array é realmente o endereço de seu primeiro elemento, imprimindo **array** e &**array [0**] usando a especificação de conversão **%p —** uma especificação especial de conversão para imprimir endereços.

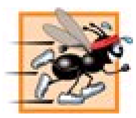
Normalmente, a especificação *%p* de conversão envia endereços como números hexadecimais. Os números hexadecimais (base 16) consistem nos dígitos 0 a 9 e as letras A a F. Eles são usados com freqüência como notação abreviada para grandes valores inteiros. O Apêndice D: Sistemas Numéricos fornece uma discussão detalhada dos relacionamentos entre os inteiros binários (base 2), octais (base 8), decimais (base 10; inteiros padrão) e hexadecimais (base 16). A saída mostra que tanto **array** como &**array [0**] possuem o mesmo valor, ou seja, **FFF0.** A saída desse programa depende do sistema, mas os endereços sempre serão idênticos.

```
/* O nome de um array e o mesmo que &array[0] */
#include <stdio.h>
main()
{
        char array[5];
        printf("    array = %p\n&array[0]  = %p\n",
                          array, &array[0]);

        return 0;
}
```

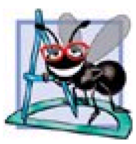
```
array = FFFO & array[0]   = FFFO
```

Fig. 6.12 O nome de um array é o mesmo que o endereço de seu primeiro elemento.



Dica de desempenho 6.4

Faz sentido passar arrays por chamadas por referência simuladas por motivos de performance. Se os arrays fossem passados por chamadas por valor, seria passada uma cópia de cada elemento. Para arrays grandes, passados freqüentemente, isso seria demorado e consumiria um armazenamento considerável para as cópias dos arrays.



Observação de engenharia de software 6.2

É possível passar um array por valor usando um truque simples explicado no Capítulo 10.

Embora arrays inteiros sejam passados simultaneamente por meio de chamadas por referência, os elementos individuais dos arrays são passados por meio de chamadas por valor exatamente como as variáveis simples. Tais porções simples de dados são chamadas *escalares* ou *quantidades escalares.* Para passar um elemento de um array para uma função, use o nome do elemento do array, com o subscrito, como argumento na chamada da função. No Capítulo 7, mostramos como simular chamadas por refência para escalares (i.e., variáveis e elementos específicos de arrays).

Para uma função receber um array por meio de uma chamada de função, sua lista de parâmetros deve especificar que um array será recebido. Por exemplo, o cabeçalho da função **modificaArray** pode ser escrito como

void modificaArray(int b[], int tamanho)

indicando que **modificaArray** espera receber um array de inteiros no parâmetro **b** e o número de elementos do array no parâmetro **tamanho.** Não é exigido o tamanho do array entre colchetes. Se ele *não* for incluído, o compilador o ignorará. Como os arrays são passados automaticamente por chamadas por referência simuladas, quando a função chamada usa o nome do array **b,** na realidade ela está se referindo ao array real no local que faz a chamada (array **temperaturaPorHora** na chamada anterior). No Capítulo 7, apresentamos outras notações para indicar que um array está sendo recebido por uma função. Como veremos, essas notações se baseiam no estreito relacionamento entre arrays e ponteiros na linguagem C.

Observe o aspecto diferente do protótipo de função para **modificaArray**

void modificaArray(int [], int);

Esse protótipo poderia ter sido escrito como

void modificaArray(int nomeQualquerArray[], int nomeQualquer Variável)

mas, como aprendemos no Capítulo 5, o compilador C ignora os nomes de variáveis em protótipos.



Boa prática de programação 6.6

Alguns programadores incluem nomes de variáveis em protótipos de funções para tornar os programas mais claros. O compilador ignora esses nomes.

Lembre-se, o protótipo diz ao compilador o número de argumentos e os tipos de cada argumento (na ordem em que os argumentos são executados) que devem aparecer.

O programa da Fig. 6.13 demonstra a diferença entre passar um array inteiro e passar um elemento de um array. O programa imprime inicialmente os cinco elementos de um array inteiro **a.** A seguir, **a** e >eu tamanho são passados para a função **modificaArray,** onde cada elemento de **a** é multiplicado por 2. Então, **a** é impresso novamente **em main.** Como mostra a saída do programa, os elementos de **a** são verdadeiramente modificados por **modificaArray.** Depois disso, o programa imprime o valor ie **a [3]** e o passa para a função **modificaElemento.** A função **modificaElemento** multiplica seu argumento por 2 e imprime o novo valor. Observe que, ao ser impresso novamente em **main, a[3]** não é modificado porque os valores individuais dos elementos do array são passados por meio de uma chamada por valor.

```
/* Passando arrays e elementos isolados de arrays para funções */
#include <stdio.h>

#define TAMANHO 5

void modificaArray(int [], int); /* parece estranho */
void modificaElemento(int);

main() {

int a[TAMANHO]   =   {0,   1,   2,   3, 4};
printf("Efeitos de passar o array inteiro por meio "
        "de chamada por referencia: \n\n OS valores do "
        "array original sao:\n");

for   (i =  0;   i <= TAMANHO -  1; i++)
          printf("%3d", a[i]);

printf("\n");

modificaArray(a, TAMANHO); /* array a passado por meio de chamada por referencia */

printf("Os valores do array modificado sao:\n");
for  (i = 0;   i <= TAMANHO -  1; i++)
          printf("%3d", a[i]);

printf("\n\nEfeitos de passar elementos do array por meio "
        "de uma chamada por valor:\n\nO valor de a[3] e %d\n", a[3]);

modificaElemento(a[3]);
printf("O valor de a[3]  e %d\n", a[3]);
return 0;
}

void modificaArray(int b[],  int tamanho){

for  (j  = 0;  j  <= tamanho - 1; j++)
          b[j]   *= 2;

}

void modificaElemento(int e){

printf("Valor em modificaElemento e %d\n",  e *= 2);

}
```

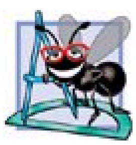
```
Efeitos de passar o array inteiro por meio de chamada por referencia:

Os valores do array original sao:
0 1 2 3 4
Os valores do array modificado sao:
0 2 4 6 8
Efeitos de passar elementos do array por meio de uma chamada por valor:
0 valor de a[3] e 6
Valor em modificaElemento e 12
O valor de a[3]  e 6
```

Fig. 6.13 Passando arrays e elementos isolados de arrays para funções.

Podem existir muitas situações em seu programa nas quais não se deve permitir que uma função modifique os elementos de um array. Como os arrays são sempre passados por chamadas por referência simuladas, as modificações nos valores de um array são difíceis de controlar. A linguagem C fornece o qualificador especial de tipo **const** para evitar a modificação dos valores de um array em uma função. Quando um parâmetro de um array é precedido pelo qualificador **const,** os elementos do array se tornam constantes no corpo da função, e qualquer tentativa de modificar ali um elemento do array resulta em um erro em tempo de compilação. Isso permite que o programador corrija um programa para que não se tente modificar o valor dos elementos do array. Embora o qualificador **const** seja bem definido no padrão ANSI, os sistemas C variam em sua capacidade de impô-lo.

A Fig. 6.14 demonstra o qualificador **const.** A função **tentaModificarArray** é definida com o parâmetro **const int b** [ ] que especifica que o array **b** é constante e não pode ser modificado. A saída mostra as mensagens de erro produzidas pelo compilador do Borland C++. Cada uma das três tentativas da função de modificar os elementos do array resulta no erro do compilador **"Cannot modify a const object. " ("Não é possível modificar um objeto const.").** O qualificador **const** será analisado novamente no Capítulo 7.



Observação de engenharia de software 6.3

*O qualificador de tipo **const** pode ser aplicado a um parâmetro de um array em uma definição de função para evitar que o array original seja modificado no corpo da função. Esse é outro exemplo do princípio do privilégio mínimo. As funções não devem ter a capacidade de modificar um array, a menos que isso seja absolutamente necessário.*

```
/* Demonstrando o qualificador do tipo const */
#include <stdio.h>

void tentaModificarArray(const int []);

main() {
```

```
int a[] = {10, 20, 30};
tentaModificarArray(a);
printf ("%d %d %d\n", a[0], a[l], a[2]);
return 0;
}

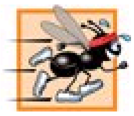
void tentaModificarArray(const int b[]) {
b[0]   /= 2; /* erro */
b[l]   /= 2; /* erro */
b[2]   /= 2; /* erro */
}
```

Fig. 6.14 Demonstrando o qualificador do tipo const.

6.6 Ordenando Arrays

Ordenar dados (i.e., colocar os dados segundo uma determinada ordem, como ascendente ou descendente) é uma das aplicações computacionais mais importantes. Um

banco classifica todos os cheques pelos números das contas para que possa preparar os extratos de cada conta no final do mês. As companhias telefônicas classificam suas listas pelo último nome e, dentro disso, pelo primeiro nome, para que fique fácil encontrar números de telefone. Praticamente todas as organizações devem classificar algum dado e, em muitos casos, quantidades muito grandes de dados. Classificar dados é um problema fascinante que tem atraído alguns dos esforços mais intensos de pesquisa no campo da ciência da computação. Neste capítulo, analisamos o que talvez seja o esquema mais simples de classificação. Nos exercícios e no Capítulo 12, investigamos esquemas mais complexos que levam a um desempenho muito superior.



Dica de desempenho 6.5

Freqüentemente, os algoritmos mais simples apresentam um desempenho muito ruim. Sua virtude reside no fato de que são fáceis de escrever, testar e depurar. Entretanto, também freqüentemente são necessários | algoritmos mais complexos para se atingir o melhor desempenho possível.

O programa da Fig. 6.15 classifica os valores dos elementos de um array **a** de dez elementos na or-dem ascendente. A técnica utilizada é chamada *classificação de bolhas (bubble sort)* ou *classificação de submersão (sinking sort)* porque os valores menores "sobem" gradualmente para o topo do array, da mesma forma que bolhas de ar sobem na água, enquanto os valores maiores afundam (submergem) para a parte de baixo do array. A técnica é passar várias vezes pelo array. Em cada passada, são comparados pares sucessivos de elementos. Se um par estiver na ordem crescente (ou se os valores forem iguais), valores são deixados como estão. Se um par estiver na ordem decrescente, seus valores são permutados no array.

Inicialmente, o programa compara **a [0**] com **a [1**], depois **a [1**] com **a [2]**, depois **a [2**] co***m*** **a [3**] e assim por diante até completar a passada comparando **a [8**] com **a [9**]. Observe que, embora existam 10 elementos, são realizadas apenas nove comparações. Tendo em vista o modo como são ***fei***-tas as comparações, um valor grande pode se mover para baixo várias posições, mas um valor pequeno só pode se mover para cima uma única posição. Na primeira passada, garante-se que o maior valor "sub-mergirá" para o elemento mais baixo (para o "fundo") do array, **a [9]**. Na segunda passada, garante-se que o segundo maior valor submergirá para **a [8**]. Na nona passada, o nono maior valor submerge para **a [1**]. Isso deixa o menor valor em **a [0**], sendo necessárias apenas nove passadas no array para classificá-lo, embora ele tenha dez elementos.

A classificação é realizada pelo loop **for** aninhado. Se for necessária uma permuta, ela é realizada pelas três atribuições seguintes

hold = a[i]; a[i]   = a[i + 1]; a[i + 1] = hold;

onde a variável extra **hold** armazena temporariamente um dos valores que está sendo permutado. A permuta não pode ser realizada com apenas as duas atribuições seguintes

a[i] = a[i + 11; a[i + 1] = a[i];

Se, por exemplo, **a[i**] é 7 e a**[i + 1**] é 5, depois da primeira atribuição ambos os valores serão 1 iguais a 5 e o valor 7 será perdido. Dessa forma, precisamos da variável extra **hold.**

A principal virtude da classificação de bolhas reside no fato de que ela é fácil de programar. Entre- j tanto, a classificação de bolhas é lenta. Isso se torna aparente durante a classificação de arrays grandes. Nos exercícios, desenvolveremos versões mais eficientes da classificação de bolhas. Classificações muito 1 mais eficientes do que a classificação de bolhas já foram desenvolvidas. Examinaremos algumas delas posteriormente no texto. Cursos mais avançados investigam classificação e pesquisa (busca) em arrays com maior profundidade.

Fig. 6.15 Classificando um array com a classificação de bolhas.

6,7 Estudo de Caso: Calculando Média, Mediana e Moda Usando Arrays

Veremos agora um exemplo maior. Os computadores são usados normalmente para compilar e analisar os resultados de pesquisas de opinião e intenções de voto. O programa da Fig. 6.16 usa o array **resposta** inicializado com 99 (representado pela constante simbólica **TAMANHO)** respostas a uma pesquisa de opinião. Cada uma das respostas é um número de 1 a 9. O programa calcula a média, a mediana e a moda dos 99 valores.

A média é a média aritmética dos 99 valores. A função **media** calcula a média somando os 99 elementos e dividindo o resultado por 99.

A mediana é o "valor do meio". A função **mediana** determina a mediana chamando a função **Bolha** para classificar o array de respostas na ordem ascendente e apanhar o elemento situado no meio do array obtido, **opiniao [TAMANHO / 2**]. Observe que, quando houver um número par de elementos, a mediana deve ser calculada como a média dos dois elementos do meio. A função **mediana** não possui atualmente a capacidade de fazer isso. A função **imprimeArray** é chamada para imprimir o array **resposta.**

A moda é o valor que ocorre mais freqüentemente entre as 99 respostas. A função **moda** determina a moda contando o número de respostas de cada tipo e depois selecionando o valor que obteve maior contagem. Essa versão da função **moda** não manipula um empate na contagem (veja o Exercício 6.14). A função **moda** produz ainda um histograma para ajudar na determinação da moda graficamente. A Fig. 6.17 contém um exemplo de execução desse programa. Esse exemplo inclui a maioria das manipulações comuns exigidas normalmente em problemas de arrays, incluindo a passagem de arrays a funções.

Fig. 6.16 Programa para análise de dados de uma pesquisa .

6.8 Pesquisando Arrays

Freqüentemente, um programador trabalhará com grandes quantidades de dados armazenadas em arrays. Pode ser necessário determinar se um array contém um valor que corresponde a um determinado *valor chave.* O processo de encontrar um determinado elemento de um array é chamado *pesquisa* (ou *busca).* Nesta seção, analisaremos duas técnicas de pesquisa — a técnica simples de *pesquisa linear* e a técnica mais eficiente de *pesquisa binária.* Os Exercícios 6.34 e 6.35 no final do capítulo pedem a implementação j de versões recursivas da pesquisa linear e da pesquisa binaria.

A pesquisa linear (Fig. 6.18) compara cada elemento do array com a *chave de pesquisa* (ou *chave de busca).* Como o array não está em uma ordem específica, as probabilidades de o valor ser encontrado no primeiro elemento ou no último são as mesmas. Entretanto, na média, o programa precisará comparar a chave de pesquisa com metade dos elementos do array.

O método de pesquisa linear funciona bem com arrays pequenos ou com arrays não-ordenados. Entretanto, para arrays grandes, a pesquisa linear é ineficiente. Se o array estiver ordenado, a técnica veloz _de pesquisa binaria pode ser utilizada.

O algoritmo de pesquisa binaria elimina metade dos elementos do array que está sendo pesquisado após cada comparação. O algoritmo localiza o elemento do meio do array e o compara com a chave de I pesquisa. Se forem iguais, a chave de pesquisa foi encontrada e o subscrito daquele elemento do array é retornado. Se não forem iguais, o problema fica reduzido a pesquisar uma metade do array. Se a chave de busca for menor que o elemento do meio do array, a primeira metade do array será pesquisada, caso contrário, a segunda metade do array será pesquisada. Se a chave de busca não for encontrada no subarray (parte do array original), o algoritmo é repetido na quarta parte do array original. A pesquisa continua até que a chave de busca seja igual ao elemento situado no meio de um subarray, ou até que o subarray consista em um elemento que não seja igual à chave de busca (i.e., a chave de busca não é encontrada).

Fig. 6.17 Exemplo de execução do programa para análise de dados de uma pesquisa.

Fig. 6.18 Pesquisa linear de um array.

No pior caso, pesquisar um array de 1024 elementos precisará de apenas 10 comparações utilizando pesquisa binaria. Dividir repetidamente 1024 por 2 leva aos valores 512, 256, 128, 64, 32, 16, 8,4, 2 e 1. O número 1024 (2^{10}) é dividido por 2 apenas 10 vezes para que seja obtido o valor 1. Dividir por 2 é equivalente a uma comparação no algoritmo de pesquisa binária. Um array com 1048576 (2^{20}) elementos precisa de um máximo de 20 comparações para que a chave de pesquisa seja encontrada. Um array com um bilhão de elementos precisa de um máximo de 30 comparações para que a chave de pesquisa seja encontrada. Isso significa um grande melhoramento de performance, comparado com a pesquisa 1 linear que exigia, em média, a comparação da chave de pesquisa com metade dos elementos do array. Para um array de um bilhão de elementos, isso significa uma diferença entre 500 milhões de comparações e um máximo de 30 comparações! As comparações máximas para qualquer array podem ser determinadas encontrando a primeira potência de 2 maior do que o número de elementos do array.

A Fig. 6.19 apresenta uma versão interativa da função **buscaBinaria.** A função recebe quatro argumentos — um array inteiro **b,** um inteiro **chaveBusca,** o subscrito **menor** do array e o subscrito **maior** do array. Se a chave de pesquisa não for igual ao elemento situado no meio de um subarray, o subscrito **menor** ou **maior** é modificado de forma que um subarray menor possa ser examinado. Se a chave de pesquisa for menor que o elemento situado no meio, o subscrito **maior** é definido como **meio - 1.** e a pesquisa continua nos elementos de **menor** a **meio - 1.** Se a chave de pesquisa for maior **do que** o elemento situado no meio, o subscrito **menor** é definido como **meio + 1,** e a pesquisa con-pÉeua nos elementos de **meio + 1** a **maior.** O programa usa um array de 15 elementos. A primeira potência de 2 maior do que o número de elementos no array é 16 (2^4), portanto, exige-se um máximo de 4 comparações para encontrar a chave de pesquisa. O programa usa a função **imprimeCabecalho | fera** exibir os subscritos do array e a função **imprimeLinha** para exibir cada subarray durante o processo da pesquisa binaria. O elemento do meio de cada subarray é marcado com um asterisco (*) para indicar o elemento com o qual a chave de busca é comparada.

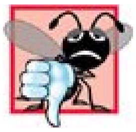
6.9 Arrays com Vários Subscritos

Os arrays em C podem ter vários subscritos. Um uso comum de arrays com vários subscritos é para representar *tabelas* de valores que consistem em informações organizadas em *linhas* e *colunas.* Para identificar um elemento específico de uma tabela, devemos especificar dois subscritos: o primeiro (por covenção) identifica a linha do elemento, e o segundo (por convenção) identifica a coluna do elemen-to. As tabelas ou arrays que exigem dois subscritos são chamados *arrays bidimensionais.* Observe que os arrays podem ter mais de dois subscritos. O padrão ANSI declara que um sistema ANSI C deve suportar pelos menos 12 subscritos de arrays.

Fig. 6.19 Pesquisa binaria de um array ordenado.

A Fig. 6.20 ilustra um array com dois subscritos, a. O array contém três linhas e quatro colunas, portanto diz-se que ele é um array 3-por-4. Em geral, diz-se que um array com *m* linhas e *n* colunas é um *array m-por-n.*

Cada elemento de **a** é identificado na Fig. 6.20 por um nome na forma **a[i] [j]**: a é o nome do ***array*** e **i** e **j** são subscritos que identificam um elemento específico em **a.** Observe que os nomes de **todos** os elementos da primeira linha têm um primeiro subscrito igual a **0;** todos os nomes dos elemen-tos da quarta coluna possuem um segundo subscrito igual a **3.**



Erro comun de programação 6.9

Fazer referência erradamente a elemento **a** *[x] [y] de um array como* **a** *[x, y].*

Um array com vários subscritos pode ser inicializado em sua declaração da mesma forma que um **aray** de apenas um subscrito *(array unidimensional).* Por exemplo, um array **b [2] [2**] com dois subscritos poderia ser declarado e inicializado com

int b[2][2] = {{1, 2), (3, 4}};

Os valores são agrupados por linha e colocados entre chaves. Assim, **1** e **2** inicializam **b[0][0**] e b**[0][1] e 3** e **4** inicializam **b[1][0]** eb**[1][1**]. Se não houver inicializadores suficientes para uma determinada linha, os elementos restantes daquela linha são inicializados com 0. Desta forma, a declaração

int b[2][2] = {{1}, {3, 4}};

inicializaria b**[0] [0]** com 1**, b[0][1]** com **0,b[1][0]** com **3 e b[1][1]** com **4.**

Fig. **6.20** Um array de dois subscritos com três linhas e quatro colunas.

A Fig. 6.21 demonstra a inicialização de arrays com dois subscritos em declarações. O programa declara três arrays de duas linhas e três colunas (seis

elementos cada). A declaração de **array1 forne** ce seis inicializadores em duas sublistas. A primeira sublista inicializa a primeira linha do array c os valores 1, 2 e 3; e a segunda sublista inicializa a segunda linha do array com os valores 4, 5 e 6. **Se** as chaves em torno de cada sublista fossem removidas da lista de inicializadores de **array1,** o com pilador inicializaria automaticamente os elementos da primeira linha e a seguir inicializaria os elemen tos da segunda linha. A declaração de **array2** fornece cinco inicializadores. Os inicializadores *são* atribuídos à primeira linha e, depois, à segunda linha. **Quaisquer** elementos que não tenham um inicializador explícito são inicializados com zero automaticamente, portanto **array2 [1][2] é** inicializado com 0. A declaração de **array3** fornece três inicializadores em duas sublistas. A sublista da primeira linha inicializa explicitamente os dois primeiros elementos da primeira linha com os valo res 1 **e** 2. O terceiro elemento é inicializado automaticamente com o valor zero. A sublista da segunda linha inicializa explicitamente o primeiro elemento com 4. Os dois últimos elementos são inicializados com zero.

O programa chama a função **imprimeArray** para criar **a** saída dos elementos de cada array. **Ob** serve que a definição da função especifica o parâmetro do array como **int a [] [3]**. Ao recebermos um array com um subscrito como argumento de uma função, as chaves ficam vazias na lista de parâmetros da função. O primeiro subscrito **de** um array com vários subscritos também não é exigido, mas todos os outros subscritos são. O compilador usa esses subscritos para determinar os locais da memória dos ele mentos de arrays com vários subscritos. Todos os elementos do array são armazenados seqüencialmente na memória, independentemente do número de subscritos. Em um array com dois subscritos, a primeira linha é armazenada na memória, seguida da segunda linha.

Fornecer os valores dos subscritos na declaração **de** parâmetros permite ao compilador informar função como localizar os elementos do array. Em um array com dois subscritos, cada linha é **basica** mente um array com um único subscrito. Para localizar um elemento de uma determinada linha, o **com** pilador deve saber exatamente quantos elementos existem em cada linha para que possa saltar o número adequado de locais da memória quando tiver acesso ao array. Desta forma, ao ter acesso a **a [1] [2} I** em nosso exemplo, o compilador sabe que deve saltar os três elementos da primeira linha na memória para obter a segunda linha (linha 1). A seguir, o compilador tem acesso ao terceiro elemento daquela linha (elemento 2).

Fig. 6.21 Inicializando arrays multidimensionais.

Muitas manipulações comuns de arrays usam estruturas de repetição for. Por exemplo, a estrutura a seguir define como zero todos os elementos da terceira linha do array a da Fig. 6.20:

```
for  (coluna = 0;  coluna <= 3; coluna++)
        a[2][coluna]   = 0;
```

Especificamos a *terceira* linha, portanto sabemos que o primeiro subscrito sempre será 2 (0 é a primeira linha, e 1 é a segunda linha). O loop for varia apenas o segundo subscrito (i.e., o subscrito da coluna). A estrutura for anterior é equivalente às instruções de atribuição:

```
a[2][0] = 0;
```

```
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

A estrutura **for** aninhada a seguir determina o total de elementos no array **a.**

```
total = 0;

for (linha = 0;  linha <= 2; linha++)
        for (coluna =0; coluna <= 3; coluna++)
                total += a[linha][coluna];
```

A estrutura **for** calcula o total de elementos do array, passando por uma linha de cada vez. A estrutura **for** externa começa definindo **linha** (i.e., o subscrito da linha) como **0** para que os elementos da pri-meira linha possam ser contados pela estrutura **for** interna. A estrutura **for** externa incrementa li-**nha** de **1,** para que os elementos da segunda linha possam ser contados. A seguir, a estrutura externa incrementa **linha** de **2,** para que os elementos da terceira linha possam ser contados. O resul-tado é impresso quando a estrutura **for** aninhada chega ao fim.

O programa da Fig. 6.22 realiza várias outras manipulações de arrays no array **grausDeAlunos** usando estruturas **for.** Cada linha do array representa um aluno e cada coluna representa um grau em um dos quatro exames que os alunos fizeram durante o semestre. As manipulações de arrays são reali-zadas por quatro funções. A função **minimo** determina o menor grau de qualquer aluno naquele semes-tre. A função **máximo** determina o maior grau de qualquer aluno no semestre. A função **media** deter-mina a média semestral de um determinado aluno. A função **imprimeArray** imprime a saída do array de dois subscritos em um formato elegante de tabela.

Cada uma das funções **minimo, máximo** e **imprimeArray** recebe três argumentos — o **grausDeAlunos** (chamado **graus** em cada função), o número de alunos (linhas do array) e o núme-ro de exames (colunas do array). Cada função executa um loop pelo array **graus** usando estruturas **for** aninhadas. A estrutura **for** aninhada a seguir é da definição da função **minimo:**

```
for  (i = 0;  i <= estud - 1; i++)
        for  (j  = 0;  j  <= testes -1; j++)
                if  (graus[i][j]  < menorGrau)
                        menorGrau = graus[i][j];
```

A estrutura **for** externa começa definindo **i** (i.e., o subscrito da linha) como **0** para que os elementos da primeira linha possam ser comparados com **menorGrau** no corpo da estrutura **for** interna. A es-trutura **for** interna percorre os quatro graus de uma determinada linha e compara cada grau com **menorGrau.** Se um grau for menor do que **menorGrau, menorGrau** é definido como esse grau. A seguir, a estrutura **for** externa incrementa o subscrito da linha em **1.** Os elementos da segunda linha são comparados com a variável **menorGrau.** A estrutura **for** externa incrementa então o subscrito da linha para **2.** Os elementos da terceira linha são comparados corn a variável **menorGrau.** Quando a execução da estrutura aninhada for concluída, **menorGrau** conterá o menor grau do array de dois subs-critos. A função **máximo** funciona de maneira similar à função **minimo.**

A função **media** utiliza dois argumentos — um array com um único subscrito de resultados de tes-tes para um estudante em particular chamado **conjDeGraus** e o número de resultados de testes no array. Quando a função **media** é chamada, o primeiro argumento passado é **grausDeAlunos [aluno].** Isso faz com que o endereço de uma linha no array de dois subscritos seja passada a **media.** O argu-mento **grausDeAlunos [1]** é o endereço inicial da segunda linha do array. Lembre-se de que **um** array de dois subscritos é basicamente um array de arrays de um único subscrito e que o nome de um array de um único subscrito é o endereço do array na memória. A função **media** calcula a soma *dos* elementos do array, divide o total pelo número de resultados de testes e retorna o valor de ponto flutuante obtido.

Fig. 6.22 Exemplo de uso de arrays de dois subscritos.

Resumo

• A linguagem C armazena em arrays listas de valores. Um array é um grupo de locais de memória relacionados entre si. Esses locais são relacionados entre si pelo fato de que todos possuem o mesma nome e o mesmo tipo. Para fazer uma referência a um local oíi elemento específico dentro do array especificamos o nome do array e o subscrito.

• Um subscrito pode ser um inteiro ou uma expressão inteira. Se um programa usa uma expressão como subscrito, a expressão é calculada para determinar o elemento específico do array.

• É importante ressaltar a diferença entre se referir ao sétimo elemento do array e o elemento sete doa array. O sétimo elemento do array tem um subscrito **6,** enquanto o elemento sete do array tem uaB subscrito **7** (na realidade, é o oitavo elemento do array). Isso é fonte de erros de "diferença-um".

• Os arrays ocupam espaço na memória. Para reservar 100 elementos para o array inteiro **b** e 27 **ele- 3** mentos para o array inteiro **x,** o programador deve escrever **int b[100], x[27];**

• Pode ser usado um array do tipo char para armazenar uma string de caracteres.

• Os elementos de um array podem ser inicializados de três maneiras: por declaração, por atribuição e por entrada de valores (input).

• Se houver menos inicializadores do que elementos do array, a linguagem C inicializa automaticamente os elementos restantes com o valor zero.

• A linguagem C não evita a referência a elementos além dos limites do array.

• Um array de caracteres pode ser inicializado por meio de uma string literal.

• Todas as strings em C terminam com o caractere nulo. A representação de constante de caracteres do caractere nulo é ' \0'.

• Os arrays de caracteres podem ser inicializados com constantes de caracteres em uma lista de icializadores.

• É possível ter acesso direto a cada caractere de uma string armazenada em um array utilizando a notação : de subscritos do array.

• Uma string pode ser fornecida diretamente em um array de caracteres a partir do teclado usando **scanf** e a especificação de conversão ***%s.***

• Um array de caracteres representando uma string pode ser impresso com **printf** e o especificador **I** de conversão ***%s.***

• Aplique **static** a uma declaração de array local para que o array não seja criado cada vez que a função for chamada e o array não seja destruído cada vez que a função for concluída.

• Os arrays declarados **static** são inicializados automaticamente uma vez em tempo de compilação. Se o programador não inicializar explicitamente um array **static,** este é inicializado com zeros pelo compilador.

• Para passar um array a uma função, o nome do array é passado. Para passar um elemento específico de um array para uma função, simplesmente passe o nome do array seguido do subscrito (entre colchetes) do elemento em particular.

• A linguagem C passa arrays para funções usando chamada por referência simulada — as funções chamadas podem modificar os valores dos elementos nos arrays originais dos locais de chamada. Na realidade, o nome do array é o endereço de seu primeiro elemento! Por ser passado o endereço inicial do array, a função chamada sabe precisamente onde o array está armazenado.

- Para receber um array como argumento, a lista de parâmetros da função deve especificar que um array será recebido. O tamanho do array não é exigido entre colchetes.
- A especificação de conversão **%p** envia endereços para o dispositivo de saída como números hexadecimais.
- A linguagem C fornece o qualificador de tipo especial **const** para evitar a modificação dos valores dos arrays em uma função. Quando um parâmetro array é precedido pelo qualificador **const,** os
elementos do array se tornam constantes no corpo da função, e qualquer tentativa de modificar um elemento ali situado resulta em um erro em tempo de compilação.
- Um array pode ser ordenado usando a técnica de classificação de bolhas (bubble sort). São feitas várias passadas do array. Em cada passada, são comparados pares sucessivos de elementos. Se um par estiver em ordem (ou se os valores forem idênticos), tudo fica como estava. Se um par estiver fora de ordem, os valores são permutados. Para arrays pequenos, a classificação de bolhas é aceitável, mas para arrays grandes ela é ineficiente, comparada com outros algoritmos de classificação mais sofisticados.
- A pesquisa (busca) linear compara cada elemento do array com a chave de busca. Como o array não se encontra em nenhuma ordem determinada, as probabilidades de o elemento ser o primeiro ou o último são as mesmas. Na média, portanto, o programa precisará comparar a chave de busca com metade dos elementos do array. O método de pesquisa linear funciona bem com arrays pequenos ou arrays que não estejam ordenados.
- O algoritmo de pesquisa (busca) binaria elimina metade dos elementos do array que está sendo pesquisado após cada comparação. O algoritmo localiza o elemento no meio do array e o compara com a chave de busca. Se forem iguais, a chave de busca foi encontrada e o subscrito do array daquele elemento é retornado. Se não forem iguais, o problema é reduzido para a pesquisa em metade do array.
- Na pior hipótese, pesquisar um array de 1024 elementos precisará de apenas 10 comparações utilizando pesquisa binária. Um array de 1048576 (2^{20}) elementos precisa de um máximo de 20 comparações para que a chave de busca seja encontrada. Um array de um bilhão de elementos precisa de
um máximo de 30 comparações para encontrar a chave de busca.
- Os arrays podem ser usados para representar tabelas de valores que consistem em informações orga nizadas em linhas c colunas. Para identificar um elemento específico de uma tabela, são especifica dos dois subscritos: o primeiro (por convenção) identifica a linha, e o segundo (por convenção) iden tifica a coluna na qual o elemento está contido. As tabelas ou arrays que exigem dois subscritos para identificar um determinado elemento são chamadas arrays bidimensionais.
- O padrão afirma que um sistema ANSI C deve suportar pelo menos 12 subscritos de arrays.
- Um array com vários subscritos pode ser inicializado em sua declaração usando uma lista de inicializadores.
- Quando recebemos um array de um único subscrito como argumento de uma função, os colchetes estão vazios na lista de parâmetros da função. O primeiro subscrito de um array com vários subscri tos também não é exigido, mas todos os subscritos subseqüentes o são. O compilador usa esses subs critos para determinar, na memória, os locais dos elementos de um array com vários subscritos.
- Para passar uma linha de um array com dois subscritos para uma função que recebe um array de um subscrito, simplesmente passe o nome do array seguido do primeiro subscrito.

Terminologia

a[i] a[i][j] análise de dados de pesquisas de opinião área temporária para permuta de valores array array bidimensional array com dois subscritos array com três subscritos array com vários subscritos array com um único subscrito array m-por-n array tridimensional array unidimensional bubble sort busca linear caractere nulo ' **\0'** chave de busca chave de pesquisa colchetes classificação classificação de bolhas classificação de submersão classificando os elementos de um array constante simbólica declarar um array diretiva de pré-processador **#define** elemento de um array elemento zero erro de "diferença-um" escalar especificação de conversão **%p** expressão como um subscrito	flexibilidade formato de tabelas gráfico de barras histograma inicializar um array lista de inicializadores do array média mediana moda nome de um array número da posição ordenando os elementos de um array passada de classificação passagem-por-referência passando arrays a funções pesquisa linear pesquisando um array precisão dupla quantidade escalar realizando uma busca em um array sair de um array sinking sort somando os elementos de um array string subscrito subscrito da coluna subscrito da linha tabela de valores texto de substituição valor de um elemento verificação dos limites

Erros Comuns de Programação

6.1 É importante observar a diferença entre o "sétimo elemento do array" e o "elemento número sete do array". Como os subscritos dos arrays iniciam em 0, o "sétimo elemento do array" tem subscrito 6, ao passo que **o** "elemento número sete do array" tem subscrito 7 e é, na realidade, o oitavo elemento do array. Esse fato **é** uma fonte de erros de "diferença-um".

6.2 Esquecer de inicializar os elementos de um array que precisam ser inicializados.

6.3 Fornecer mais inicializadores para um array do que o número de seus elementos é um erro de sintaxe.

6.4 Encerrar uma declaração **#define** ou #**include** com ponto-e-vírgula. Lembre-

se de que as diretivas do pré-processador não são instruções da linguagem C.

6.5 Atribuir um valor a uma constante simbólica em uma instrução executável é um erro de sintaxe. Uma constante simbólica não é uma variável. O compilador não reserva espaço para ela, como faz com as variáveis que possuem valores em tempo de execução.

6.6 Fazer referência a um elemento além dos limites de um array.

6.7 Não fornecer a **scanf** um array de caracteres com comprimento suficiente para armazenar uma string digitada no teclado pode resultar em perda de dados de um programa e outros erros em tempo de execução.

6.8 Considerar que os elementos de um array local declarado **static** são inicializados com zero sempre que a função na qual o array é declarado for chamada.

6.9 Fazer referência erradamente a elemento **a [x] [y]** de um array como **a [x, y].**

Práticas Recomendáveis de Programação

6.1 Use apenas letras maiúsculas para nomes de constantes simbólicas. Isso faz essas constantes se destacarem no programa e lembra o programador que as constantes simbólicas não são variáveis.

6.2 Procure obter clareza nos programas. Algumas vezes vale a pena prejudicar o uso mais eficiente da memória visando a tornar programas mais claros.

6.3 Ao fazer um loop em um array, os subscritos nunca devem ser menores do que 0 e sempre devem ser menores do que o número total de elementos do array (tamanho - 1). Certifique-se de que a condição de término do loop evita o acesso a elementos fora desses limites.

6.4 Mencione o maior subscrito de um array em uma estrutura **for** para ajudar a eliminar erros de diferença-um.

6.5 Os programas devem verificar a exatidão de todos os valores de entrada para evitar que informações inadequadas afetem os cálculos ali realizados.

6.6 Alguns programadores incluem nomes de variáveis em protótipos de funções para tornar os programas mais claros. O compilador ignora esses nomes.

Dicas de Performance

6.1 Algumas vezes as considerações de desempenho são muito mais importantes do que as considerações de clareza.

6.2 Os efeitos (geralmente graves) da referência a elementos situados fora dos limites de um array dependem dos sistemas.

6.3 Em funções que contêm arrays automáticos onde se entra e sai do escopo daquela função freqüentemente, faça com que o array seja **static,** para que ele não seja criado sempre que a função for chamada.

6.4 Faz sentido passar arrays por chamadas por referência simuladas por motivos de desempenho. Se os arrays fossem passados por chamadas por valor, seria passada uma cópia de cada elemento. Para arrays grandes, passados freqüentemente, isso seria demorado e consumiria um armazenamento considerável para as cópias dos arrays.

6.5 Freqüentemente, os algoritmos mais simples apresentam um desempenho muito ruim. Sua virtude reside no fato de que são fáceis de escrever, testar e depurar. Entretanto, também freqüentemente são necessários algoritmos mais complexos para se atingir o melhor desempenho possível.

Observações de Engenharia de Software

6.1 Definir o tamanho de cada array como uma constante simbólica torna os programas mais flexíveis.
6.2 É possível passar um array por valor usando um truque simples explicado no Capítulo 10.
6.3 O qualificador de tipo **const** pode ser aplicado a um parâmetro de um array em uma definição de função para evitar que o array original seja modificado no corpo da função. Esse é outro exemplo do princípio do privilégio mínimo. As funções não devem ter a capacidade de modificar um array, a menos que isso seja absolutamente necessário.

Exercícios de Revisão

6.1 Complete as lacunas:

a) Listas e tabelas de valores são armazenadas em_.
b) Os elementos de um array são relacionados entre si pelo fato de que possuem o mesmo ____________ e _______________.
c) O número usado para fazer referência a um elemento específico de um array é chamado seu __________
d) Deve ser usado um(a) ________________ para declarar o tamanho de um array porque isso torna os programas mais flexíveis.
e) O processo de colocar em ordem os elementos de um array é chamado _________ um array.
f) O processo de determinar se um array contém um valor chave específico é chamado ________________ o array
g) Um array que usa dois subscritos é chamado array ______________ .

6.2 Diga se cada uma das sentenças a seguir é verdadeira ou falsa. Se a resposta for falsa, explique o motivo

a) Um array pode armazenar muitos tipos diferentes de valores.
b) Um subscrito de array pode ser um tipo de dado **float.**
c) Se houver menos inicializadores em uma lista do que o número de elementos no array, a lingua gem C inicializa automaticamente os elementos restantes com o último valor da lista de inicializa
dores.
d) É um erro se uma lista de inicializadores possuir mais inicializadores do que o número de elementos do array.
e) Um elemento em particular de um array que for passado a uma função e modificado na função chamada terá seu valor modificado na função que chamou.

6.3 Responda às seguintes perguntas a respeito de um array chamado **fracoes.**

a) Defina uma constante simbólica TAMANHO a ser substituída pelo texto de substituição 10.
b) Declare um array com tantos elementos do tipo **float** quanto o valor de TAMANHO e inicialize os elementos com o valor **0.**
c) Dê o nome do quarto elemento a partir do início do array.

d) Chame o elemento 4 do array.
e) Atribua o valor **1. 667** ao elemento nove do array.
f) Atribua o valor **3.333** ao sétimo elemento do array.
g) Imprima os elementos 6 a 9 do array com dois dígitos de precisão à direita do ponto decimal e mostre a saída que é realmente apresentada na tela.
h) Imprima todos os elementos do array usando uma estrutura de repetição **for.** Admita que a variável in teira **x** foi definida como uma variável de controle para o loop. Mostre a saída do programa.

6.4 Responda às seguintes perguntas a respeito de um array chamado **tabela.**

a) Declare o array como um array inteiro com 3 linhas e 3 colunas. Considere que a constante simbólica TAMANHO foi definida como 3.
b) Quantos elementos o array contém?
c) Use uma estrutura de repetição **for** para inicializar cada elemento do array com a soma de seus subscri tos. Presuma que as variáveis inteiras **x** e **y** são declaradas como variáveis de controle.
d) Imprima os valores de cada elemento do array **tabela.** Presuma que o array foi inicializado com a de claração,
int tabela[TAMANHO][TAMANHO] = **{{1, 8}, {2, 4, 6}, {5}};**
e as variáveis inteiras **x** e **y** são declaradas como variáveis de controle. Mostre a saída.

6.5 Encontre o erro em cada um dos segmentos de programa a seguir e o corrija.
a) **#define** TAMANHO **100;**
b) TAMANHO = **10;**
c) Assuma **int b[10] = {0}, i;**
for (i = 0; i <= 10; i++)
d) **#include <stdio.h**>**;**
e) Assuma int **a[2][2J = {{1, 2}, {3, 4}}; a[1,1] = 5;**

Respostas dos Exercícios de Revisão

6.1 a) Arrays. b) Nome, tipo. c) Subscrito, d) Constante simbólica, e) Classificação, f) Pesquisa, g) Bidimensional

6.2 a) Falso. Um array pode armazenar apenas valores do mesmo tipo.
b) Falso. Um subscrito de array deve ser um inteiro ou uma expressão inteira.
c) Falso. A linguagem C inicializa automaticamente os elementos restantes com zeros.
d) Verdadeiro.
e) Falso. Os elementos isolados de um array são passados por meio de chamadas por valor. Se o array intei ro for passado a uma função, quaisquer modificações se refletirão no original.

6.3 a) **#define TAMANHO 10**
b) **float fracoes[TAMANHO] = {0};**
c) **fracoes [3]**
d) **fracoes[4]**
e) **fracoes [9] = 1.667;**
f) **fracoes [6] = 3.333;**
g) **printf ("%.2'f %.2f\n", fracoes[6], fracoes[9]);** *Saída:* **3.33 1.67.**
h) for (x = 0; X <= **TAMANHO - 1;** X++)
printf("fracoes*[%d]* = %f\n", x, fracoes[x]);
Saída:
fracoes[0] = 0.000000 fracoes[1] = 0.000000 fracoes[2] = 0.000000 fracoes[3] = 0.000000 fracoes[4] = 0.000000 fracoes[5] = 0.000000 fracoes[6] = 3.333000 fracoes[7] = 0.000000 fracoes[8] = 0.000000 fracoes[9] = 1.667000 6.4

6.4 a) **int tabela [TAMANHO] [TAMANHO] ;**
b) Nove elementos.
c) **for (x = 0;** X <= **TAMANHO - 1;** X++)
for (y = 0; y <= TAMANHO - 1; y++) tabela[x][y] = x + y;
d) for (x = 0; x <= TAMANHO - 1; X+ +)
for (y = 0; y <= TAMANHO - 1; y++) printf("tabela[*%d][%d]* =%d\n", x, y, tabela[x][y]);
Saída:
tabela[0][0] = 1
tabela[0][1] = 8
tabela[0][2] = 0
tabela[1][0] = 2
tabela[1][1] = 4
tabela[1][2] = 6
tabela[2][0] = 5
tabela[2][1] = 0
tabela[2][2] = 0

6.5 a) Erro: Ponto-e-vírgula no final da diretiva de pré-processador **#define.**
Correção: Eliminar o ponto-e-vírgula.
b) Erro: Atribuir um valor a uma constante simbólica usando uma instrução de atribuição.
Correção: Atribuir um valor a uma constante simbólica em uma diretiva de pré-

processador **#define** sem usar o operador de atribuição, como em **#define TAMANHO 10.**
c) Erro: Fazer referência a um elemento além dos limites do array **(b [10]).** Correção: Mudar o valor final da variável de controle para **9.**
d) Erro: Ponto-e-vírgula no final da diretiva de pré-processador **#include.** Correção: Eliminar o ponto-e-vírgula.
e) Erro: Subscritos do array de forma incorreta. Correção: Mudar a instrução para **a [1][1] = 5 ;**

Exercícios

6.6 Preencha as lacunas em cada uma das sentenças a seguir:
a) A linguagem C armazena listas de valores em _____________.
b) Os elementos de um array estão relacionados entre si pelo fato de que __________.
c) Ao se referir a um elemento de array, o número da posição contida entre colchetes é chamado _______________.
d) Os nomes dos cinco elementos do array **p** são ______, ____________, ___________. ____________ e ______________
e) O conteúdo de um elemento determinado do array é chamado _____________ daquele elemento.
f) Denominar um array, declarar seu tipo e especificar seu número de elementos é chamado_______________ um array.
g) O processo de colocar elementos de um array em ordem descendente ou ascendente é chamado ________________ .
h) Em um array bidimensional, o primeiro subscrito (por convenção) identifica a ___________________ de um elemento, e o segundo subscrito (por convenção) identifica a _______________ de um elemento.
i) Um array m-por-n contém ____________ linhas, ________ colunas e _____________ elementos.
j) O nome do elemento na linha 3 e coluna 5 do array **d** é ________________ .

6.7 Diga quais das sentenças a seguir são verdadeiras e quais são falsas; explique os motivos pelos quais as sentenças foram consideradas falsas.
a) Para se referir a um local ou um elemento em particular dentro de um array, especificamos o nome de array e o valor daquele elemento.
b) Uma declaração de array reserva espaço para ele.
c) Para indicar que 100 locais devem ser reservados para um array inteiro **p,** o programador deve escrever a declaração
p[100] ;
d) Um programa em C que inicializa todos os 15 elementos de um array com zeros deve conter uma instrução **for.**
e) Um programa em C que soma os elementos de um array de dois subscritos deve conter instruções aninhadas.
f) A média, a mediana e a moda do seguinte conjunto de valores são 5, 6 e 7 respectivamente: 1, 2, 5,*6*,*7*,7,7.

6.8 Escreva instruções em C que realizem o seguinte:
a) Apresentem na tela o valor do sétimo elemento do array de caracteres **f.**
b) Entrem com um valor no elemento 4 de um array unidimensional de ponto flutuante **b.**
c) Inicializem cada um dos 5 elementos de um array unidimensional inteiro **g** com o valor **8.**

d) Somem os elementos de um array de ponto flutuante **c** que possui 100 elementos.
e) Copiem o array **a** para a primeira parte do array **b.** Considere **float a [11], b[34] ;**
f) Determine e imprima o menor e o maior valor contidos em um array de ponto flutuante **w** com 99 elementos.

6.9 Considere um array inteiro 2-por-5 **t.**
a) Escreva uma declaração para **t.**
b) Quantas linhas **t** possui?
c) Quantas colunas **t** possui?
d) Quantos elementos **t** possui?
e) Escreva os nomes de todos os elementos da segunda linha de **t.**
f) Escreva os nomes de todos os elementos da terceira coluna de **t.**
g) Escreva uma única instrução em C que defina como zero o elemento da linha 1 e coluna 2 de **t.**
h) Escreva uma série de instruções em C que inicialize como zero cada elemento de **t.** Não use uma estrutura de repetição.
i) Escreva uma estrutura **for** aninhada que inicialize cada elemento de **t** como zero.
j) Escreva uma instrução em C que entre com os valores dos elementos de **t** a partir do terminal.
k) Escreva uma série de instruções em C que determine e imprima o menor valor do array **t.**
1) Escreva uma instrução em C que exiba na tela os elementos da primeira linha de **t.**
m) Escreva uma instrução em C que some os elementos da quarta coluna de **t.**
n) Escreva uma série de instruções em C que imprima o array **t** em um formato organizado de tabela. Liste os subscritos das colunas como cabeçalho ao longo do topo e liste os subscritos das linhas à esquerda de cada linha.

6.10 Use um array unidimensional para resolver o seguinte problema. Uma companhia paga seus vendedores com base em comissões. O vendedor recebe $200 por semana mais 9 por cento de suas vendas brutas daquela semana. Por exemplo, um vendedor que teve vendas brutas de $3000 em uma semana recebe $200 mais 9 por cento de $3000, ou seja, um total de $470. Escreva um programa em C (usando um array de contadores) que determine quantos vendedores receberam salários nos seguintes intervalos de valores (considere que o salário de cada vendedor é trancado para que seja obtido um valor inteiro):
1. $200-$299
2. $300-$399
3. $400-$499
4. $500-$599
5. $600-$699
6. $700-$799
7. S800-S899
8. $900-$999
9. $1000 em diante

6.11 A classificação de bolhas (bubble sort) apresentada na Fig. 6.15 é ineficiente para arrays grandes. Faça as modificações simples a seguir para melhorar o desempenho da classificação de bolhas.

a) Depois da primeira passada, garante-se que o maior número está no elemento de maior número do array; depois da segunda passada, os dois maiores números estão em

"seus lugares", e assim por diante. Em vez, de fazer nove comparações en cada passada, modifique a classificação de bolhas para fazer oito comparações na segunda passada, sete na terceira e assim por diante.
b) Os dados do array já podem estar na ordem adequada ou quase ordenados completamente, assim, por que fazer nove passadas se um número menor seria suficiente? Modifique a classificação para verificar se alguma permuta foi feita ao final de cada passada. Se nenhuma permuta foi realizada, os dados já devem estar na ordem adequada e o programa deve ser encerrado. Se foram feitas permutas, pelo menos uma passada é necessária.

6.12 Escreva instruções simples que realizem cada uma das operações seguintes em um array unidimensional:
a) Inicialize com zeros os 10 elementos do array inteiro **contagem.**
b) Adicione 1 a cada um dos 15 elementos do array inteiro **bonus.**
c) Leia os 12 valores do array de ponto flutuante **temperaturasMensais** a partir do teclado.
d) Imprima os 5 valores do array inteiro **melhoresEscores** em um formato de coluna.

6.13 Encontre o(s) erro(s) em cada uma das seguintes instruções:
a) Considere: **char str[5];**
scanf("%s", str); /* Usuário digita hello */
b) Considere: **int a[3] ;**
printf("$d *%d %d\n"*, a[1], a[2], a[3]);
c) **float f[3] = {1.1, 10.01, 100.001, 1000.0001};**
d) Considere: **double d[2] [10] ; d[l, 9] = 2.345;**

6.14 Modifique o programa da Fig. 6.16 de forma que a função **moda** seja capaz de manipular um empate no valor da moda. Modifique também a função **media** de forma que seja encontrada a média dos dois valores do meio de um array com número par de elementos.

6.15 Use um array unidimensional para resolver o seguinte problema. Leia 20 números, todos situados entre 10 e 100, inclusive. À medida que cada número for lido, imprima-o somente se não for duplicata de um número já lido. Experimente o "pior caso", no qual todos os 20 números são diferentes. Use o menor array possível para resolver esse problema.

6.16 Classifique os elementos do array bidimensional 3-por-5 **vendas** para indicar a ordem na qual eles são definidos como zero pelo seguinte segmento de programa:
for (linha = 0; linha <= 2; linha++)
for (coluna = 0; coluna <= 4; coluna++) vendas[linha][coluna] = 0; 6.17

6.17 O que faz o seguinte programa? **#include <stdio.h> #define TAMANHO 10**
int quelssoFaz(int [], int};
main()
int total, a[TAMANHO] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; total = quelssoFaz(a, TAMANHO);
printf("Soma dos valores dos elementos do array e *%d\n"*, total); return 0;
}
int quelssoFaz(int b[], int tamanho) {
if (tamanho == 1)

return b[0]; else
return b[tamanho - 1] + quelssoFaz(b, tamanho - 1);
}

6.18 O que faz o seguinte programa? **#include <stdio.h> #define TAMANHO 10**
void algumaFuncao(int [], int};
main()
int a[TAMANHO] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37);
printf("Os valores no array sao: \n"); algumaFuncao(a, TAMANHO);
printf("\n"); return 0;
void algumaFuncao(int b[], int tamanho)
if (tamanho > 0) {
algumaFuncao(&b[1], tamanho - 1); printf*("%d* ", b[0]);

6.19 Escreva um programa em C que simule a jogada de dois dados. O programa deve usar **rand** para rolar o primeiro dado e deve usar **rand** novamente para rolar o segundo dado. A soma dos dois valores deve então ser calculada. *Nota:* Como cada dado pode mostrar um valor inteiro de 1 a 6, a soma dos dois valores \
de 2 a 12 com 7 sendo a soma mais freqüente e 2 e 12 sendo as somas menos freqüentes. A Fig. 6.23 mostra as 36 combinações possíveis dos dois dados. Seu programa deve rolar os dados 36.000 vezes. Use um array unidimensional para registrar o número de vezes que cada soma possível é obtida. Imprima **os** resultados em um formato de tabela. Além disso, determine se as somas são razoáveis, i.e., há seis maneiras de obter 7, portanto, aproximadamente um sexto do total de jogadas deve ser 7.

6.20 Escreva um programa que rode 1000 jogos de Craps e responda a cada uma das seguintes perguntas:
a) Quantos jogos são vencidos na primeira jogada, segunda jogada,vigésima jogada e depois da vigésima **jogada?**
b) Quantos jogos são perdidos na primeira jogada, segunda jogada,vigésima jogada e depois da vigésima jogada?
c) Quais as probabilidades de vencer no jogo de Craps? *(Nota:* Você deve descobrir que Craps é um jogos de cassino mais honestos. O que você acha que isso significa?)
d) Qual a duração média de um jogo de Craps?
e) As probabilidades de vencer aumentam com a duração do jogo?

6.21 *(Sistema de Reserva Aérea)* Uma pequena companhia aérea acabou de comprar um computador para o seu novo sistema automático de reservas. O presidente pediu a você que programasse o novo sistema em C. Você deve escrever um programa para atribuir assentos a cada vôo do único avião da companhia (capacidade: 10 assentos). Seu programa deve exibir o seguinte menu de alternativas:

Favor digitar 1 para "fumante"
Favor digitar 2 para "naofumante"

Se a pessoa digitar 1, seu programa deve fazer a reserva de um assento no setor dos fumantes (assentos 1-5). Se a pessoa digitar 2, seu programa deve reservar um assento no setor de não-fumantes (assentos 6-10).

	1	2	3	4	5	6
1	**2**	**3**	**4**	**5**	**6**	**7**
2	**3**	**4**	**5**	**6**	**7**	**8**
3	**4**	**5**	**6**	**7**	**8**	**9**
4	**5**	**6**	**7**	**8**	**9**	**10**
5	**6**	**7**	**8**	**9**	**10**	**11**
6	**7**	**8**	**9**	**10**	**11**	**12**

Fig. 6.23 Os 36 resultados possíveis da jogada de dois dados.

Seu programa deve então imprimir um cartão de embarque indicando o número do assento do passageiro e se ele se encontra no setor de fumantes ou de não-fumantes do avião.

Use um array unidimensional para representar o esquema dos assentos do avião. Inicialize todos os elementos do array com 0 para indicar que todos os assentos estão livres. A medida que cada assento for reservado, iguale os elementos correspondentes a 1 para indicar que o assento não está mais disponível.

Seu programa nunca deve, obviamente, reservar um assento que já tenha sido distribuído. Quando o setor de fumantes estiver lotado, seu programa deve perguntar se a pessoa aceita um lugar no setor de não-fumantes (e vice-versa). Em caso positivo, faça a reserva apropriada do assento. Em caso negativo, imprima a mensagem **"Próximo voo sai em 3 horas."**

6.22 Use um array bidimensional para resolver o seguinte problema. Uma companhia tem quatro vendedores (1 a 4) que vendem 5 produtos diferentes (1 a 5). Uma vez por dia, cada vendedor elabora um memorando de cada tipo diferente de produto vendido. Cada memorando contém:

1. O número do vendedor
2. O número do produto
3. O valor total em dólares daquele produto vendido naquele dia

Dessa forma, cada vendedor elabora de 0 a 5 memorandos de vendas por dia. Admita que as informações de todos os memorandos do último mês estão disponíveis. Escreva um programa que leia todas essas informaçõess das vendas do último mês e faça um resumo do total de vendas por vendedor e por produto. Todos os totais devem ser armazenados no array bidimensional **vendas.** Depois de processar todas as informações do último mês, imprima os resultados em um formato de tabela com cada coluna representando um vendedor diferente e cada linha representando um produto diferente. Faça a soma dos valores de cada linha para obter as vendas totais de cada produto no último mês; faça a soma dos valores em cada coluna para obter as vendas totais de cada vendedor no último mês. A tabela impressa deve incluir esses totais à direita das linhas somadas e abaixo das colunas somadas.

6.23 *(Gráfico da Tartaruga).* A linguagem Logo, que é particularmente popular entre os usuários de computadores pessoais, tornou famoso o conceito dos *gráficos da tartaruga.* Imagine uma tartaruga mecânica que percorra uma sala sob o controle de um programa em C. A tartaruga possui uma caneta em uma de duas posições, para cima ou para baixo. Quando a caneta está para baixo, a tartaruga desenha figuras à medida que se move; quando a caneta está para cima, a tartaruga se move livremente sem desenhar nada. Neste problema, você simulará a operação da tartaruga e criará também um esquema computadorizado de movimento.

Use um array **plano** 50-por-50 inicializado com zeros. Leia comandos de um array que os contém. Controle sempre a posição atual da tartaruga e se a caneta está para

cima ou para baixo. Admita que a tartaruga sempre começa na posição 0,0 do plano com sua caneta para cima. O conjunto de comandos da tartaruga que seu programa deve processar são os seguintes:

Comando	Significado
1	Caneta para cima
2	Caneta para baixo
3	Giro para direita
4	Giro para a esquerda
5, 10	Movimento 10 espaços a frente (ou outro número diferente de 10
6	Imprime o array 50-por-50
9	Fim dos dados (sentinela)

Suponha que a tartaruga esteja em algum lugar próximo do centro do plano. O "programa" a seguir desenharia e imprimiria um quadrado 12-por-12:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
 6
 9
```

À medida que a tartaruga se move com a caneta para baixo, defina os elementos apropriados do array plano como 1s. Quando o comando 6 (imprimir) for emitido, onde houver um 1 no array exiba um asterisco ou outro caractere de sua escolha. Sempre que houver um zero, exiba um espaço em branco. Escreva programa em C para implementar os recursos do gráfico da tartaruga descritos aqui. Escreva vários programas de gráficos da tartaruga para desenhar formas interessantes. Adicione outros comandos para aumentar a potencialidade de sua linguagem de gráfico de tartaruga.

6.24 *(Passeio do Cavalo)* Um dos problemas mais interessantes para os fãs do jogo de xadrez é o problema do Passeio do Cavalo, proposto originalmente pelo matemático Euler. A questão é esta: a peça do jogo de drez chamada cavalo pode se mover ao longo de um tabuleiro vazio e passar uma única vez em cada uma 64 casas? Estudamos aqui esse interessante problema em profundidade.

O cavalo faz movimentos em formato de L (percorre duas casas em uma direção e uma no sentido perpendicular às duas primeiras). Dessa forma, de uma casa no meio do tabuleiro, o cavalo pode fazer movimentos diferentes (numerados de 0 a 7), como mostra a Fig. 6.24.

a) Desenhe um tabuleiro de xadrez 8-por-8 em uma folha de papel e tente fazer o Passeio do Cavalo a mão, Coloque um 1 no primeira casa para a qual se mover, um 2 na segunda casa, um 3 na terceira etc. **Antes** de começar os movimentos, imagine até onde você chegará, lembrando-se que um passeio completo consiste em 64 movimentos. Até onde você chegou? Você chegou perto do quanto havia imaginado?

	0	1	2	3	4	5	6	7
0								
1				**2**		**1**		
2			**3**				**0**	
3					**K**			
4			**4**				**7**	
5				**5**		**6**		
6								
7								

Fig. 6.24 Os oito movimentos possíveis do cavalo,

b) Agora vamos desenvolver um programa que moverá o cavalo pelo tabuleiro de xadrez. O tabuleiro em si é representado pelo array bidimensional 8-por-8 tabuleiro. Cada um dos quadrados (casas) é inicializado com o valor zero. Descrevemos cada um dos oito movimentos possíveis em termos de seus componentes horizontais e verticais. Por exemplo, um movimento do tipo 0. como mostra a Fig. 6.24, consiste em mover dois quadrados horizontalmente para a direita e um quadrado verticalmente para cima. O movimento 2 consiste em mover um quadrado horizontalmente para a esquerda e dois quadrados verticalmente para cima. Os movimentos horizontais para a esquerda e verticais para cima são indicados por números negativos. Os oito movimentos podem ser descritos por arrays bidimensionais, **horizontal** e **vertical,** como **se** segue:

horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal [6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1

As variáveis **linhaAtual** e **colunaAtual** indicam a linha **e** a coluna da posição atual do cavalo. Para fazer um movimento do tipo **numMov,** em que **numMov** está entre 0 e 7, seu programa usa as instruções

```
linhaAtual += vertical[numMov];
colunaAtual += horizontal[numMov];
```

Utilize um contador que varie de **1** a **64.** Grave a última contagem em cada casa para a

qual o cavalo se mover. Lembre-se de testar cada movimento potencial para verificar **se** cavalo já esteve naquele quadrado. E, obviamente, teste cada movimento potencial para **se** certificar de que o cavalo não está saindo do tabuleiro. Agora escreva um programa para mover o cavalo pelo tabuleiro de xadrez. Rode o programa. Quantos movimentos o cavalo fez? c) Depois de tentar escrever e rodar o programa do Passeio do Cavalo, provavelmente você adquiriu alguns conceitos valiosos. Usaremos **esses** conceitos para desenvolver uma *heurística* (ou estratégia) para mover o cavalo. A heurística não garante o sucesso, mas uma heurística desenvolvida cuidadosamente aumenta as probabilidades de ele ser atingido. Você pode ter observado que os quadrados externos são, de alguma forma, mais problemáticos do que os quadrados próximos ao centro do tabuleiro. Na realidade, os quadrados mais problemáticos, ou inacessíveis, são os quatro cantos.

A intuição pode sugerir que você deve tentar mover o cavalo para os quadrados mais problemáticos e deixar livres os quadrados mais fáceis de atingir para que, quando o tabuleiro ficar congestionado próximo ao final do passeio, haja maior probabilidade de sucesso.

Podemos desenvolver uma "heurística de acessibilidade" classificando cada um dos quadrados de acordo com sua acessibilidade e depois sempre mover o cavalo para o quadrado (dentro dos movimentos em forma de L, obviamente) que seja mais inacessível. Colocamos no array bidimensional **acessibilidade** os números que indicam a partir de quantos quadrados um determinado quadrado pode ser acessado. Portanto, em um tabuleiro vazio, as casas centrais são classificadas com **8s, os** quadrados dos cantos são classificados com **2s** e **os** outros quadrados possuem números de acessibilidade **3, 4** e **6,** como é mostrado a seguir

2	**3**	**4**	**4**	**4**	**4**	**3**	**2**
3	**4**	**6**	**6**	**6**	**6**	**4**	**3**
4	**6**	**8**	**8**	**8**	**8**	**6**	**4**
4	**6**	**8**	**8**	**8**	**8**	**6**	**4**
4	**6**	**8**	**8**	**8**	**8**	**6**	**4**
4	**6**	**8**	**8**	**8**	**8**	**6**	**4**
3	**4**	**6**	**6**	**6**	**6**	**4**	**3**
2	**3**	**4**	**4**	**4**	**4**	**3**	**2**

Agora escreva uma versão do programa do Passeio do Cavalo usando a heurística de acessibilidade. Em qualquer tempo, o cavalo deve se mover para o quadrado com menor número de acessibilidade. Em caso de empate, o cavalo pode **se** mover para qualquer dos quadrados que empataram. Portanto, o passeio pode começar em qualquer um dos quatro cantos. *(Nota:* A medida que o cavalo se mover no tabuleiro, seu programa deve reduzir os números de acessibilidade quando cada vez mais quadrados forem ocupados. Dessa forma, a qualquer instante durante o passeio o número de acessibilidade de cada quadrado disponível permanecerá igual ao número de quadrados dos quais aquele espaço pode ser alcançado.) Rode essa versão de seu programa. Você conseguiu fazer um passeio completo? Agora modifique o programa para executar 64 passeios, um para cada quadrado do tabuleiro. Quantos passeios completos você coseguiria fazer?

d) Escreva uma versão do programa do Passeio do Cavalo na qual, ao encontrar um empate entre dois ou mais quadrados, você faça sua escolha verificando os quadrados que podem ser alcançados a partir dos quadrados "empatados". Seu programa deve fazer o movimento do cavalo para o quadrado para o qual o próximo movimento levar ao quadrado com menor número de acessibilidade.

6.25 *(Passeio do Cavalo: Métodos de Força Bruta)* No Exercício 6.24 desenvolvemos uma solução para **o pro**blema do Passeio do Cavalo. O método usado, chamado "heurística de acessibilidade", gera muitas **soluções** e funciona de uma maneira eficiente.
Como os computadores continuam a ser cada vez mais poderosos, poderemos resolver muitos problemas com base apenas no poder computacional e em algoritmos relativamente simples. Vamos chamar esse método de resolução de problemas de "força bruta".
a) Use a geração de números aleatórios para permitir ao cavalo percorrer o tabuleiro (em seus movimentos permitidos na forma de L, obviamente) de maneira aleatória. Seu programa deve executar um passeio e imprimir o tabuleiro final. Até onde o cavalo chegou?
b) Muito provavelmente, o programa anterior levará a um passeio relativamente curto. Agora modifique seu programa para tentar 1000 passeios. Use um array unidimensional para controlar o número de passeios por extensão alcançada. Quando seu programa terminar de fazer os 1000 passeios, ele deve imprimir essas informações em um formato de tabela. Qual o melhor resultado?
c) Muito provavelmente, o programa anterior forneceu alguns passeios "respeitáveis" mas nenhum passeio completo. Agora "retire os limites" e simplesmente deixe seu programa ser executado até que um passeio completo seja produzido. *(Cuidado:* Essa versão do programa pode rodar durante horas em um computador poderoso.) Mais uma vez. conserve urna tabela do número de passeios para cada extensão alcançada e imprima essa tabela quando o primeiro passeio completo for realizado. Quantos passeios seu programa tentou fazer antes de produzir um passeio completo? Quanto tempo levou?
d) Compare a versão de força bruta do Passeio do Cavalo com a versão de heurística de acessibilidade. Qual exigiu um estudo mais cuidadoso do problema? Que algoritmo foi mais difícil de desenvolver? Qual exige mais poder computacional? Poderíamos estar certos (antecipadamente) de obter um passeio completo cea| a heurística de acessibilidade? Poderíamos estar certos (antecipadamente) de obter um passeio completo com o método da força bruta? Analise as vantagens e desvantagens da resolução de problemas em geral por força bruta.

6.26 *(Oito Damas)* Outro quebra-cabeça para os fãs do jogo de xadrez é o problema das Oito Damas. Resumidamente: é possível colocar oito damas em um tabuleiro vazio de xadrez, de forma que nenhuma dama "ataque" qualquer uma das outras, isto é, de forma que não haja duas damas na mesma linha, na mesma coluna ou ao longo da mesma diagonal? Use o método de raciocínio desenvolvido no Exercício 6.24 para formular uma heurística para resolver o problema das Oito Damas. Execute seu programa. *(Dica:* É possível ai
um valor numérico a cada quadrado do tabuleiro indicando quantos quadrados são "eliminados" se uma for colocada ali. Por exemplo, cada um dos quatro cantos receberia o valor 22, como na Fig. 6.25.)
Depois de esses "números de eliminações" serem colocados em todas as 64 casas (quadrados), uma heurística apropriada poderia ser: coloque a próxima dama no quadrado com menor número de eliminações Por que essa estratégia é intuitivamente atraente?

6.27 *(Oito Damas: Métodos de Força Bruta)* Neste problema, você desenvolverá vários métodos de força bruta para resolver o problema das Oito Damas apresentado no Exercício 6.26.

```
********
**
* *
*  *
*   *
*    *
*     *
*      *
```

Fig. 6.25 Os 22 quadrados (casas) eliminados ao colocar uma dama no canto superior esquerdo do tabuleiro.

a) Resolva o problema das Oito Damas usando a técnica de força bruta aleatória desenvolvida no Exercício 6.25.
b) Use uma técnica completa, i.e., tente todas as combinações possíveis das oito damas no tabuleiro.
c) Por que você acha que o método da força bruta pode não ser apropriado para resolver o problema das Oito Damas?
d) Compare e observe as diferenças entre os métodos da força bruta aleatória e da força bruta completa em geral.

6.28 *(Eliminação de Valores Duplicados)* No Capítulo 12 exploramos a estrutura de dados da árvore de pesquisa binaria de alta velocidade. Um recurso de uma árvore de pesquisa binária é que os valores duplicados são eliminados ao serem feitas inserções na árvore. Isso é chamado eliminação de valores duplicados. Escreva um programa que produza 20 números aleatórios entre 1 e 20. O programa deve armazenar todos os valores não-duplicados em um array. Use o menor array possível para realizar essa tarefa.

6.29 *(Passeio do Cavalo: Teste do Passeio Fechado)* No Passeio do Cavalo, um passeio completo é aquele em que o cavalo faz 64 movimentos passando por cada casa (quadrado) do tabuleiro apenas uma vez. Um passeio fechado ocorre quando o movimento 64 está a um movimento do local onde o cavalo começou o passeio. Modifique o programa do Passeio do Cavalo escrito no Exercício 6.24 para verificar se um passeio completo obtido é um passeio fechado.

6.30 *(A Peneira de Eratóstenes)* Um inteiro primo é qualquer inteiro que só pode ser dividido exatamente por si mesmo e por 1. A Peneira de Eratóstenes é um método de encontrar números primos. Ele funciona da seguinte maneira:
1) Crie um array com todos os elementos inicializados com 1 (verdadeiro). Os elementos do array com subscritos primos permanecerão 1. Todos os outros elementos do array serão definidos posteriormente como zero.
2) Começando com o subscrito 2 do array (o subscrito 1 deve ser primo), sempre que for encontrado um elemento do array cujo valor seja 1, faça um loop pelo restante do array e defina como zero todos os elementos cujo subscrito seja múltiplo do subscrito com valor 1. Para o subscrito 2 do array, todos os elementos além do 2 que forem múltiplos de 2 serão definidos como zero (subscritos 4, 6, 8, 10 etc). Para o subscrito 3 do array, todos os elementos além do 3 que forem múltiplos de 3 serão definidos como zero (subscritos 6, 9, 12, 15 etc).
Quando esse processo for concluído, os elementos do array que ainda estiverem definidos como 1 indicam que o subscrito é um número primo. Esses subscritos

podem ser impressos. Escreva um programa que use um array de 1000 elementos para determinar e imprimir os números primos entre 1 e 999. Ignore o elemento 0 do array.

6.31 *(Classificação de Depósitos)* Uma classificação de depósitos começa com um array unidimensional de inteiros positivos a ser classificado e um array bidimensional de inteiros com linhas possuindo subscritos de 0 a 9 e colunas com subscritos de 0 a n - 1, onde *n é o* número de valores do array a ser classificado. Cada linha do array bidimensional é chamada um depósito (bucket). Escreva uma função **bucketClass** que utilize um array inteiro e o tamanho do array como argumentos. O algoritmo é o seguinte:

1) Faça um loop pelo array unidimensional e coloque cada um de seus valores em uma linha do array de depósitos, baseado nos dígitos na casa das unidades. Por exemplo, 97 é colocado na linha 7. 3 é colocado na linha 3 e 100 é colocado na linha 0.

2) Faça um loop pelo array de depósitos e copie os valores de volta para o array original. A próxima ordem dos valores acima no array unidimensional será 100, 3 e 97.

3) Repita esse processo para posição subseqüente dos dígitos (dezenas, centenas, milhares etc.) e pare quando o dígito da extremidade esquerda do maior número for processado.

Na segunda passada do array, 100 é colocado na linha 0, 3 é colocado na linha 0 (possui apenas um dígito) e 97 é colocado na linha 9. A ordem dos valores no array unidimensional é 100, 3 e 97. Na terceira passada, 100 é colocado na linha 1, 3 é colocado na linha zero e 97 é colocado na linha zero (depois do 3). A classificação de depósito garante a colocação de todos os valores em uma ordem apropriada após o processamento do dígito mais à esquerda do maior número. A classificação de depósito sabe que chegou ao fim quando todos os valores forem copiados na linha zero do array bidimensional.

Observe que o array bidimensional de depósitos é dez vezes maior que o tamanho do array inteiro a ser ordenado. Essa técnica de ordenação (classificação) fornece um desempenho melhor do que a classificação de bolhas, mas exige capacidade de armazenamento muito maior. A classificação de bolhas exige apenas um local adicional na memória para o tipo de dados a ser classificado. A classificação de depósitos é um exemplo de troca de espaço por tempo. Ela usa mais memória, mas tem melhor desempenho. Essa versão de classificação de depósitos exige a cópia de todos os dados novamente no array original ao fim de cada passada. Outra possibilidade é criar um segundo array bidimensional de depósitos e mover repetidamente os dados entre os dois arrays de depósitos até que todos os dados estejam copiados na linha zero de um dos arrays. A linha zero conterá então o array ordenado.

Exercícios de Recursão

6.32 *(Classificação de Seleção)* Uma classificação de seleção pesquisa um array à procura de seu menor elemento. Quando o menor elemento for encontrado, ele é permutado com o primeiro elemento do array. O processo é então repetido para o subarray que se inicia com o segundo elemento do array. Cada passada do **array** resulta na colocação de um elemento em seu local apropriado. Essa classificação exige capacidade de processamento idêntica à da classificação de bolhas — para um array de *n* elementos, devem ser feitas *n - 1* passadas, e para cada subarray devem ser feitas *n* - 1 comparações para encontrar o menor valor. Quando o subarray a ser processado possuir um elemento, o array estará ordenado. Escreva uma função recurisva **seleClass** para realizar esse algoritmo.

6.33 *(Palíndromos)* Um palíndromo é uma string que é soletrada da mesma forma da frente para trás ou de trás para a frente. Alguns exemplos de palíndromos são "radar" "orava o avaro" e "socorram marrocos". Escreva uma função recursiva **testePalind** que retorne 1 se a string armazenada no array for um palíndromo, ou 0 em caso contrário. A função deve ignorar espaços e pontuação na string.

6.34 *(Pesquisa Linear)* Modifique o programa da Fig. 6.18 para usar uma função recursiva **pesqLinear para** realizar a pesquisa linear no array. A função deve receber um array inteiro e o tamanho do array como argumentos. Se a chave de busca (pesquisa) for encontrada, retorne o subscrito do array; caso contrário, retorne -1.

6.35 *(Pesquisa Binária)* Modifique o programa da Fig. 6.19 para usar uma função recursiva **pesqBinar para** realizar a pesquisa binária no array. A função deve receber um array inteiro e os subscritos inicial e final como argumentos. Se a chave de busca (pesquisa) for encontrada, retorne o subscrito do array; caso contrário, retorne — 1.

6.36 *(Oito Damas)* Modifique o programa Oito Damas criado no Exercício 6.26 para que o problema seja resolvido recursivamente.

6.37 *(Imprimir um Array)* Escreva uma função recursiva **imprimeArray** que tome um array e seu tamanho. como argumentos, imprima o array e não retorne nada. A função deve parar o processamento e retornar quando receber um array de tamanho zero.

6.38 *(Imprimir uma string de trás para a frente)* Escreva uma função recursiva **inverteString** que tome um array de caracteres como argumento, imprima o array de caracteres na ordem inversa e nada retorne. A função deve parar o processamento e retornar quando um caractere nulo de terminação de string for encontrado.

6.39 *(Encontrar o valor mínimo de um array)* Escreva uma função recursiva **miniRecursivo** que tome um array inteiro e o tamanho do array como argumentos e retorne o menor elemento do array. A função deve parar o processamento e retornar quando receber um array de 1 elemento.

7

Ponteiros

Objetivos

- Ser capaz de usar ponteiros.
- Ser capaz de usar ponteiros para passar argumentos a funções chamadas por referência.
- Entender os estreitos relacionamentos entre ponteiros, arrays e strings. - Entender o uso de ponteiros para funções.
- Ser capaz de declarar e usar arrays de strings.

Endererereços nos são dados para ocultar nossos paradeiros.
(H.H. Munro)

Por referências indiretas, descubra os rumos a seguir.
William Shakespeare
Hamlet

Muitas coisas, tendo referência total Para alguém, podem ser contraditórias
William Shakespeare
Henrique V

Você verá que é um hábito muito saudável verificar sempre suas referências, siri
Dr. Routh

Você não pode confiar em um código que não criou completamente. Especialmente o código de companhias que empregam pessoas como eu.)
Ken Thompson
Palestra no Turing Award de 1983 Association for Computing Machinery, Inc.

Sumário

7.1 Introdução

Neste capítulo, analisaremos um dos recursos mais poderosos da linguagem de programação C, o pon*teiro.* Os ponteiros estão entre os aspectos do C mais difíceis de dominar. Os ponteiros permite programas simular chamadas por referência e criar e manipular estruturas dinâmicas de dados, i.e, estruturas de dados que podem crescer e diminuir, como as listas encadeadas, filas (queues), pilhas (stacks) e árvores (trees). Este capítulo explica os conceitos básicos de ponteiros. O Capítulo 10 examina o uso de ponteiros com estruturas. O Capítulo 12 apresenta as técnicas de gerenciamento dinâmico da memória e mostra exemplos de criação e uso de estruturas dinâmicas de dados.

7.2 Declarações e Inicialização de Variáveis Ponteiros

Os ponteiros são variáveis que contêm endereços de memória como valores. Normalmente, uma variavel faz uma referência direta a um valor específico. Um ponteiro, por outro lado, contém um endereço de uma variável que contém um valor específico. Sob esse ponto de vista, um nome de variável faz uma referência *direta* a um valor, e um ponteiro faz referência *indireta* a um valor (Fig. 7.1). Diz-se que fazer referência a um valor por meio de um ponteiro é fazer uma *referência indireta.*

Os ponteiros, como quaisquer outras variáveis, devem ser declarados antes de serem usados.

int *contPtr, cont;

A declara a variável **contPtr** do tipo **int** * (i.e.. um ponteiro para um valor inteiro) e é lida como **"contPtr** é um ponteiro para **int''** ou **"contPtr** aponta para um objeto do tipo inteiro [integer]". Além disso, a variável **cont** é declarada como inteira e não como um ponteiro para um valor inteiro. O * se aplica somente a **contPtr** na declaração. Quando o * é usado dessa forma em uma declaração, ele indica que a variável que está sendo declarada é um ponteiro. Os ponteiros podem ser declarados para apontar objetos de qualquer tipo de dados.

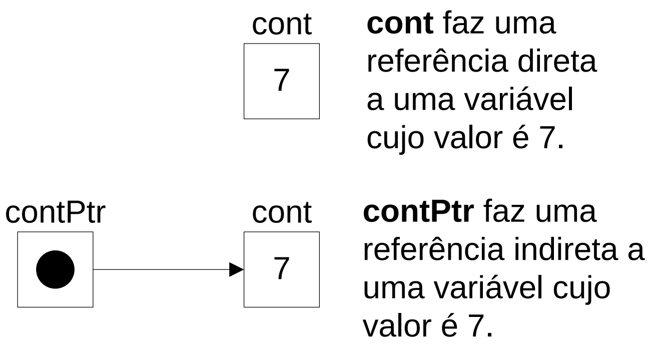
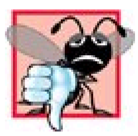


Fig 7.1 Fazendo referência direta e indireta a uma variável.



Erro comun de programação 7.1

*O operador de referência indireta * não se aplica a iodos os nomes de variáveis em uma declaração. Cada Ponteiro deve ser declarado com o * colocado antes do nome.*



Boa prática de programação 7.1

Incluir as letras ptr em nomes de variáveis de ponteiros para tornar claro que essas variáveis são ponteiros e precisam ser manipuladas apropriadamente.

Os ponteiros devem ser inicializados ao serem declarados ou em uma instrução de atribuição. Um ponteiro pode ser inicializado com 0, **NULL** ou um endereço. Um ponteiro com o valor **NULL** não aponta para lugar algum. **NULL** é uma constante

simbólica definida no arquivo de cabeçalho **<stdio. h>** (e em vários outros arquivos de cabeçalho). Inicializar um ponteiro com 0 é equivalente a inicializar um ponteiro com NULL mas **NULL** é mais recomendado. Quando 0 é atribuído, inicialmente ele é convertido em um ponteiro do tipo apropriado. O valor 0 é o único valor inteiro que pode ser atribuído diretamente a uma variável de ponteiro. Atribuir um endereço de variável a um ponteiro é analisado na Seção 7.3.



Boa prática de programação 7.2

Inicialize ponteiros para evitar resultados inesperados.

7.3 Operadores de Ponteiros

O & ou *operador de ponteiro é* um operador unário que retorna o endereço de seu operando. Por exemplo, admitindo as declarações

int y = 5;
int *yPtr;

a instrução

yPtr = &y;

atribui o endereço da variável **y** à variável de ponteiro **yPtr.** Diz-se que a variável **yPtr** "aponta para" **y.** A Fig. 7.2 mostra uma representação esquemática da memória depois que a atribuição anterior é executada.

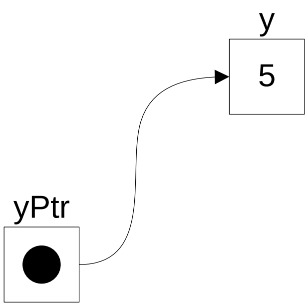


Fig. 7.2 Representação gráfica de um ponteiro apontando para uma variável inteira na memória.



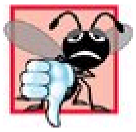
Fig. 7.3 Representação de **y** e **yPtr** na memória.

A Fig. 7.3 mostra a representação do ponteiro na memória, admitindo que a variável de ponteiro y está armazenada no local 600000ea variável de ponteiro yPtr está armazenada no local 50000. O operando do operador de endereço deve ser uma variável; o operador de endereço não pode ser aplicado a constantes, expressões ou a variáveis declaradas com a classe de armazenamento register.

O operador *, chamado freqüentemente *operador de referência indireta* ou *operador de desreferenciamento,* retorna o valor do objeto ao qual seu operando (i.e., um ponteiro) aponta. Por exemplo, a instrução

printf("%d", *yPtr);

imprime o valor da variável y, ou seja, 5. Usar * dessa maneira é chamado *desreferenciar um poteiro.*



Erro comun de programação 7.2

Desreferenciar um ponteiro que não foi devidamente inicializado ou que não foi atribuído para apontar para um local específico da memória. Isso poderia causar um erro fatal de tempo de execução, ou poderia modificar acidentalmente dados importantes e permitir que o programa seja executado até o final fornecendo resultados incorretos.

```
/* Usando os operadores & e * */
#include <stdio.h>
main() {

int a; /* a e um inteiro */
int *aPtr; /* aPtr e um ponteiro para um inteiro */
a = 7;
aPtr = &a; /* aPtr define o endereço de a */

printf("0 endereço de a e %p\n"
        "O valor de aPtr e %p\n\n", &a, aPtr);

printf("O valor de a e %d\n"
        "O valor de *aPtr e %d\n\n", a, *aPtr);

printf("Sabendo que * e & complementam-se mutuamente."
        "\n&*aPtr = %p\n*&aPtr = %p\n",&*aPtr,  *&aPtr);

return 0;
}
```

```
O endereço de a e FFF4
O valor de aPtr e FFF4
O valor de a e 7
O valor de *aPtr e 7
Sabendo que * e & complementam-se mutuamente.
&*aPtr = FFF4
*&aPtr = FFF4
```

Fig. 7.4 Os operadores & e *,

<table>
<tr><th>Operadores</th><th>Associatividade</th><th>Tipo</th></tr>
<tr><td>( ) [ ]</td><td>Da esquerda para direita</td><td>maior</td></tr>
<tr><td>++ -- ! (tipo)</td><td>Da direita para esquerda</td><td>Unário</td></tr>
<tr><td>* / %</td><td>Da esquerda para direita</td><td>Multiplicativo</td></tr>
<tr><td>+ -</td><td>Da esquerda para direita</td><td>Aditivo</td></tr>
<tr><td>< <= > >=</td><td>Da esquerda para direita</td><td>Relacional</td></tr>
<tr><td>== !=</td><td>Da esquerda para direita</td><td>Igualdade</td></tr>
<tr><td>&&</td><td>Da esquerda para direita</td><td>E lógico</td></tr>
<tr><td>| |</td><td>Da esquerda para direita</td><td>Ou lógico</td></tr>
<tr><td>?:</td><td>Da direita para esquerda</td><td>Condicional</td></tr>
<tr><td>= += -= *= /= %=</td><td>Da direita para esquerda</td><td>Atribuição</td></tr>
<tr><td>,</td><td>Da esquerda para direita</td><td>Vírgula</td></tr>
</table>

Fig. 7.5 Precedência dos operadores.

O programa da Fig. 7.4 demonstra os operadores de ponteiros. A especificação de conversão %p envia *para* o dispositivo de saída o local da memória como inteiro hexadecimal (veja o Apêndice D, Sistemas Numéricos, para obter mais informações sobre inteiros hexadecimais). Observe que o endereço do valor de aPtr são idênticos na saída, confirmando assim que o endereço de a é realmente atribuído à variável de ponteiro aPtr. Os operadores & e * complementam-se mutuamente — quando os dois forem aplicados consecutivamente a aPtr em qualquer ordem, os mesmos resultados são impressos. *A t*abela da Fig. 7.5 mostra a precedência e a associatividade dos operadores apresentados até agora.

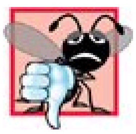
7.4 Chamando Funções por Referência

Há duas maneiras de passar argumentos a uma função — chamada por valor e chamada por referência.

Todas as chamadas de funções em C são chamadas por valor. Como vimos no Capítulo 5, return pode ser usado para retornar um valor de uma função chamada para o local que a chamou (ou retornar controle de uma função chamada sem passar um valor de volta). Muitas funções exigem a capacidade de modificar uma ou mais variáveis no local chamador, ou passar um ponteiro para um grande objeto de dados para evitar o overhead de passar o objeto chamado por valor (que, obviamente, exige fazer uma cópia do objeto). Para essa finalidade, a linguagem C fornece a capacidade de simular chamadas por referência.

Em C, os programadores usam ponteiros e o operador de referência indireta para simular chamadas por referência. Ao chamar uma função com argumentos que devem ser modificados, são passados os endereços dos argumentos. Isso é realizado normalmente aplicando o operador de endereço (&) à variavel cujo valor será modificado. Como vimos no Capítulo 6, os arrays não são passados usando o operador & porque a linguagem C passa automaticamente o local inicial do array na memória (o nome do array é equivalente a &nomeArray [0]). Quando o endereço de uma variável é passado a uma função, o operador de referência indireta (*) pode ser usado na função para modificar o valor no local de memória do chamador. Os programas das Figs. 7.6 e 7.7 apresentam duas versões de uma função que eleva um inteiro ao cubo — cuboPorValor e cuboPorReferencia. O programa da Fig. 7.6 passa a variável numero para a função cuboPorValor usando uma chamada por valor. A função cuboPorValor eleva ao cubo seu argumento e passa o novo valor de volta para main usando uma instrução return. O novo valor é atribuído a numero em main.

O programa da Fig. 7.7 passa a variável numero usando uma chamada por referência — o endereço de numero é passado — para a função cuboPorReferencia. A função cuboPorReferencia toma o ponteiro para int chamado nPtr como argumento. A função desreferencia o ponteiro e eleva ao cubo o valor para o qual nPtr aponta. Isso muda o valor de numero em main. As Figs. 7.8 e 7.9 analisam graficamente os programas das Figs. 7.6 e 7.7, respectivamente.



Erro comun de programação 7.3

Não desreferenciar um ponteiro quando necessário para obter o valor para o qual o ponteiro aponta.

```c
/* Eleva uma variável ao cubo usando chamada por valor */
#include <stdio.h>
int cuboPorValor(int);
main() {
int numero = 5;
printf("0 valor original do numero e %d\n", numero);
numero = cuboPorValor(numero);
printf("0 novo valor do numero e %d\n", numero);
return 0;
}

int cuboPorValor(int n)
{
return n * n * n;      /* eleva ao cubo a variável local n */
}
```

```
O valor original do numero e 5
O novo valor do numero e 125
```

Fig. 7.6 Elevar uma variável ao cubo usando uma chamada por valor.

```c
/* Eleva uma variável ao cubo usando chamada por referência */
#include <stdio.h>
int cuboPorValor(int);
main() {
int numero = 5;
printf("0 valor original do numero e %d\n", numero);
numero = cuboPorValor(&numero);
printf("0 novo valor do numero e %d\n", numero);
return 0;
}

void cuboPorValor(int *nPtr)
{
*nPtr  = *nPtr * *nPtr * *nPtr;      /* eleva ao cubo a variável local n */
}
```

```
O valor original do numero e 5
O novo valor do numero e 125
```

Fig. 7.7 Elevar uma variável ao cubo usando uma chamada por referência.

Uma função que recebe um endereço como argumento deve definir um parâmetro de ponteiro para receber o endereço. Por exemplo, o cabeçalho da função cuboPorReferencia é

void cuboPorReferencia(int *nPtr)

O cabeçalho especifica que cuboPorReferencia recebe o endereço de uma variável inteira **como** argumento, armazena o endereço localmente em nPtr e não retorna um valor.

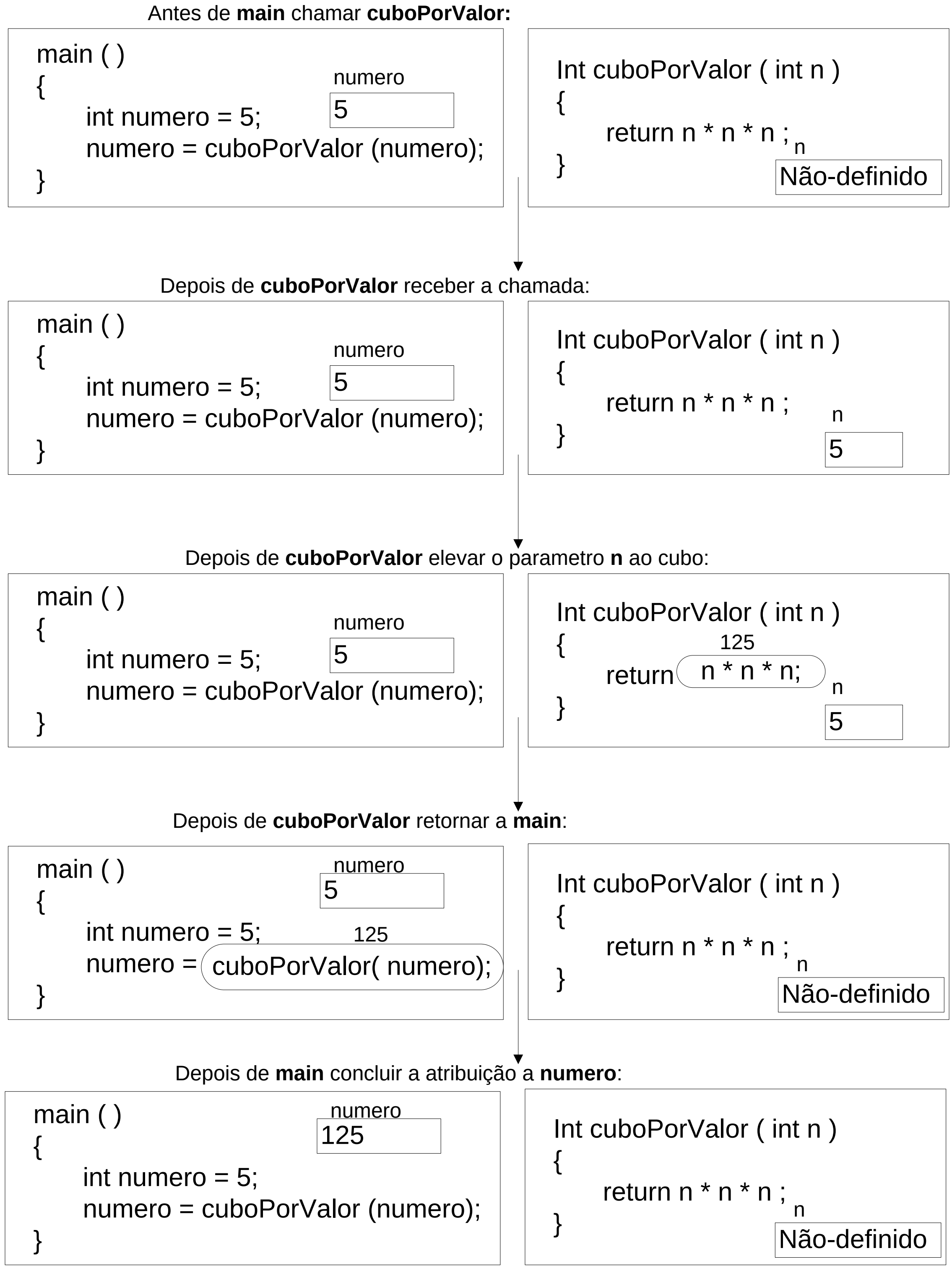


Fig. 7.8 Análise de uma típica chamada por valor.

O protótipo de função para **cuboPorRef erencia** contém **int** * entre parênteses. Da mesma forma que com outros tipos de variáveis, não se faz necessário incluir nomes de ponteiros em protótipos de funções. Os nomes incluídos com a finalidade de servir de documentação são ignorados pelo Compilador C.

No cabeçalho da função e no protótipo de uma função que espera um array unidimensional como argumento, pode ser utilizada a notação de ponteiros na lista de parâmetros de **cuboPorReferencia.** O compilador não faz diferença entre uma função que recebe um ponteiro e uma função que recebe um array b unidimensional. Isso, obviamente, significa que a função deve "saber" quando está recebendo um array ou simplesmente uma variável única para a qual deve realizar uma chamada por referência. Quando o compilador encontra um parâmetro de função para um array unidimensional na forma **int b** [ ], o compilador converte o parâmetro para a notação de ponteiros **int *b.** As duas formas são intercambiáveis.

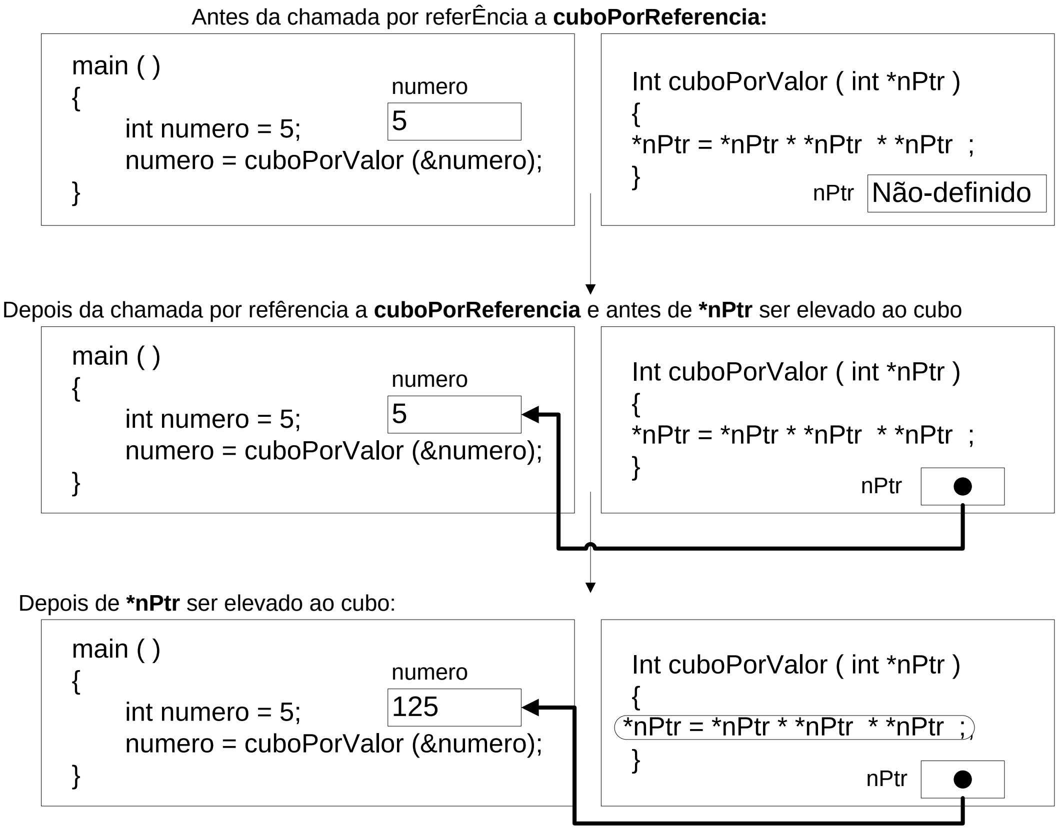


Fig. 7.9 Análise de uma típica chamada por referência.

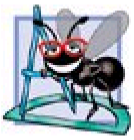


Boa prática de programação 7.3

*Use uma chamada por valor para passar argumentos a uma função, a menos que o local chamador **explicite** que a função chamada modifique o valor da variável do argumento no ambiente original. Isso é outro **exemplo** do princípio do privilégio mínimo.*

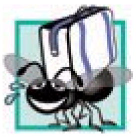
7.5 Usando o Qualificador Const com Ponteiros

O *qualificador const* permite ao programador informar ao compilador que o valor de uma determina da variável não deve ser modificado. O qualificador **const** não existia nas primeiras versões do C; ele foi adicionado à linguagem pelo comitê ANSI C.



Observação de engenharia de software 7.1

*O qualificador **const** pode ser usado para impor o princípio do privilégio mínimo. Usar o principio do privilégio mínimo para desenvolver software diminui tremendamente o tempo de depuração e efeitos colaterais indesejáveis e torna um programa mais fácil de modificar e manter.*



Dicas de portabilidade 7.1

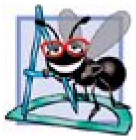
*Embora **const** seja bem definido no ANSI C, alguns sistemas não o impõem.*

Com o passar dos anos, uma grande base de código legado foi escrito nas primeiras versões do C que **não** usavam **const** porque esse qualificador não estava disponível. Por esse motivo, há enormes oportunidades de melhoria na engenharia de software do antigo código C. Além disso, muitos programadores que usam atualmente o ANSI C não utilizam **const** em seus programas porque começaram a programar empregando as primeiras versões do C. Esses programadores estão perdendo muitas oportunidades de praticar uma boa engenharia de software.

Existem seis possibilidades de usar (ou não) **const** com parâmetros de funções — duas com a passagem de parâmetros por meio de uma chamada por valor e quatro por meio de uma chamada por referência. Como escolher uma das seis possibilidades? Deixe que o princípio do privilégio mínimo seja seu guia. Sempre ceda *a u*ma função acesso suficiente aos dados em seus parâmetros para realizar sua tarefa específica, não mais.

No Capítulo 5, explicamos que todas as chamadas em C são chamadas por valor — é feita uma cópia do argumento na chamada da função e esta cópia é passada à função. Se a cópia for modificada na função, o valor original é mantido sem modificação no local que executou a chamada. Em muitos casos, um valor passado a uma função é modificado para que a função possa realizar sua tarefa. Entretanto, algumas vezes, o valor não deve ser alterado na função chamada, muito embora a função manipule uma cópia do valor original.

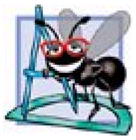
Considere uma função que utiliza um array unidimensional e seu tamanho como argumentos e imprima o array. Tal função deve fazer um loop através do array e enviar para impressão cada elemento do array individualmente. O tamanho do array é usado no corpo da função para determinar o maior subscrito do array de forma que o loop possa terminar quando a impressão for concluída. O tamanho do array não se modifica no corpo da função.



Observação de engenharia de software 7.2

Se um valor não se modifica (ou não deve ser modificado) no corpo de uma função à qual é passado, ele **deve** *ser declarado* ***const*** *para evitar que seja modificado acidentalmente.*

Se for feita uma tentativa de modificar um valor declarado **const,** o compilador a reconhece e emite um aviso ou um erro, dependendo do compilador em particular.



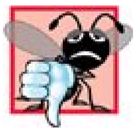
Observação de engenharia de software 7.3

Apenas um valor pode ser alterado em uma função chamada por valor. Esse valor deve ser atribuído pelo valor de *retorno da função. Para modificar vários valores em uma função, deve-se utilizar chamada por referência.*



Boa prática de programação 7.4

Antes de usar uma função, verifique o seu protótipo para determinar se ela é capaz de modificar os valores passados a ela.



Erro comun de programação 7.4

Não saber que uma função está aguardando ponteiros como argumentos de uma chamada por referência e passar argumentos de chamada por valor. Alguns compiladores tomam os valores admitindo que são ponteiros e os desreferenciam como tais. Em tempo de execução, são geradas violações de acesso à memória ou falhas de segmentação. Outros compiladores captam incompatibilidade de tipos entre os argumentos e parametros e geram mensagens de erro.

Há quatro maneiras de passar um ponteiro para uma função: um ponteiro não-constante para um dado não-constante, um ponteiro constante para um dado não-constante, um ponteiro não-constante para um dado constante e um ponteiro constante para um dado constante. Cada uma das quatro combinações fornece um nível diferente de privilégios de acesso.

O maior nível de acesso aos dados é concedido por um ponteiro não-constante cara um dado não-constante. Nesse caso, o dado pode ser modificado por meio de um ponteiro desreferenciado, e o ponteiro pode ser modificado de modo a apontar para outros itens de dados. Uma declaração de um ponteiro não-constante para um dado não-constante não inclui **const.** Tal ponteiro pode ser usado para receber uma string como um argumento de uma função que use a aritmética dos ponteiros para processar (e possivelmente modificar) cada caractere da string. A função **converteParaMaiusculas** da Fig. 7.10 declara como argumento um ponteiro não-constante para dados não-constantes chamado **s (char *s).** A função processa um caractere da string **s** por vez usando a aritmética de ponteiros. Se um caractere estiver no intervalo de **a** a **z,** ele é convertido para sua letra maiúscula correspondente **A** a **z** usando um cálculo baseado em seu código ASCII; caso contrário, o caractere é ignorado e o próximo caractere da

string é processado. Observe que todas as letras maiúsculas do conjunto de caracteres ASCII possuem valores inteiros que são equivalentes ao valor de suas letras minúsculas correspondentes menos 32 (veja o Apêndice C para obter uma tabela ASCII de valores dos caracteres). No Capítulo 8, apresentamos a função **toupper** da biblioteca padrão do C para converter letras minúsculas em maiúsculas.

Um ponteiro não-constante para um dado constante é um ponteiro que pode ser modificado para apontar para qualquer item de dado do tipo apropriado, mas o dado ao qual ele aponta não pode ser modificado. Tal ponteiro pode ser usado para receber um argumento array em uma função que processará cada elemento do array sem modificar os dados. Por exemplo, a função **imprimeCaracteres** da Fig. *7.11* declara serem do tipo **const char** * os parâmetros **s.** A declaração é lida da direita para a esquerda como **"s** é um ponteiro para um caractere constante". O corpo da função usa uma estrutura **for para** enviar ao dispositivo de saída cada caractere da string até que o caractere null seja encontrado. Depois de cada caractere ser impresso, o ponteiro **s** é incrementado de modo a apontar para o próximo caractere na string.

```
/* Converter letras minúsculas para maiúsculas */
/* usando um ponteiro nao-constante para um dado nao-constante */
#include <stdio.h>

void converteParaMaiusculas(char *);

main() {
char string[ ]  = "caracteres";
printf("A string antes da conversão e: %s\n", string);
converteParaMaiusculas(string);
printf("A string depois da conversão e: %s\n", string);
return 0;
}

void converteParaMaiusculas(char *s) {
while  (*s !=   '\0') {
        if  (*s>='a' && *s<='z')
                *s -= 32; /* converte para a letra maiúscula ASCII */
        ++s;      /* incrementa s para apontar para o próximo caractere */
}
}
```

A string antes da conversão e: caracteres
A string depois da conversão e: CARACTERES

Fig. 7.10 Convertendo uma string para letras maiúsculas usando um ponteiro não-constante para um dado não-constante.

```
/* Imprimir um caractere de uma string de cada vez */
/* usando um ponteiro nao-constante para um dado constante */
#include <stdio.h>

void imprimeCaracteres(const char *);

main(){
char string [ ]  = "imprime caracteres de uma string";

printf("A string e:\n");
imprimeCaracteres(string);
putchar('\n');
return 0;
}

void imprimeCaracteres (const char *s) {
for ( ; *s !=  '\0' ; s++)       /* nenhuma inicialização */
          putchar(*s);
}
```

```
A string e:
imprime caracteres de uma string
```

Fig. 7.11 Imprimindo um caractere de uma string de cada vez usando um ponteiro não-constante para um dado constante

```
/* Tentando modificar dados por meio de um */
/* ponteiro nao-constante para dados constantes */
#include <stdio.h>

void f(const int *);

main ( ) {
int y;
f(&y); /* f tenta uma modificação ilegal */
return 0;
}

void f(const int *x) {

*x = 100; /* impossivel modificar um objeto const */

}
```

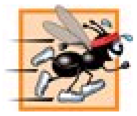
```
Compiling FIG7_12.C:
Error FIG7_12.C 17: Cannot modify a const object
Warning FIG7_12.C 18: Parameter 'x' is never used
```

Fig. 7.12 Tentando modificar dados por meio de um ponteiro não-constante para dados constantes.

A Fig. 7.12 mostra as mensagens de erro produzidas pelo compilador Borland C++ ao tentar compilar uma função que recebe um ponteiro não-constante para um dado constante e que usa o ponteiro para modificar o dado.

Como sabemos, os arrays são tipos agregados de dados que armazenam muitos itens de dados relacionados nados entre si, do mesmo tipo, sob um mesmo nome. No Capítulo 10, analisaremos outra forma de um tipo agregado de dados chamado *estrutura* (chamado algumas vezes *registro* em outras linguagens).

Uma estrutura é capaz de armazenar muitos itens de dados relacionados entre si, de diferentes tipos dados, sob um mesmo nome (e.g., armazenar informações sobre cada empregado de uma companhia. Quando uma função é chamada tendo um array como argumento, o array é passado automaticamente para a função por meio de uma chamada por referência. Entretanto, as estruturas são sempre passadas por meio de uma chamada por valor — é passada uma cópia de toda a estrutura. Isso exige o overhead em tempo de execução de fazer uma cópia de cada item dos dados na estrutura e armazená-lo na pilha de chamada da função no computador. Quando a estrutura de dados deve ser passada a uma função. podemos usar ponteiros para dados constantes para obter o desempenho de uma chamada por referência com a proteção de uma chamada por valor. Quando é passado um ponteiro a uma estrutura, deve ser feita apenas uma cópia do endereço no qual a estrutura está armazenada. Em um equipamento com endereços de 4 bytes, é feita uma cópia de 4 bytes de memória em vez de uma cópia de possivelmente centenas ou milhares de bytes da estrutura.



Dica de desempenho 7.1

Passe objetos grandes como estruturas usando ponteiros para dados constantes para obter as vantagens de desempenho da chamada por referência e a segurança da chamada por valor.

Usar ponteiros para dados constantes dessa maneira é um exemplo de *compensação de tempo/espaço.* Se a memória estiver pequena e a eficiência da execução for a principal preocupação, devem ser usados ponteiros. Se houver memória em abundância e a eficiência não for a preocupação principal, os dados devem ser passados por meio de chamadas por valor para impor o princípio do privilégio mínimo. Lembre-se de que alguns sistemas também não trabalham com **const,** portanto a chamada por valor ainda é a melhor maneira de evitar que os dados sejam modificados.

Um ponteiro constante para um dado não-constante é um ponteiro que sempre aponta para o mesmo local da memória, e os dados naquele local podem ser modificados por meio do ponteiro. Esse é o default para um nome de array. Um nome de array é um ponteiro constante para o início do array. Pode-se ter acesso a todos os dados do array e modificá-los usando o nome do array e os subscritos. Um ponteiro constante para dados não-constantes pode ser usado para receber um array como argumento de uma função que tenha acesso aos elementos do array usando apenas a notação de seus subscritos. Os ponteiros declarados const devem ser inicializados ao serem declarados (se o ponteiro for um parâmetro da função, ele é inicializado com um ponteiro que é passado à função). O programa da Fig. 7.13 tenta modifi car um ponteiro constante. O ponteiro **ptr** é declarado do tipo **int * const.** A declaração é lida da direita para a esquerda como **"ptr** é um ponteiro constante para um inteiro". O ponteiro é inicializado com o

endereço da variável inteira **x.** O programa tenta atribuir o endereço de **y** a **ptr,** mas uma mensagem de erro é gerada.

```
/* Tentando modificar um ponteiro constante para */
*/ dados nao-constantes */
#include <stdio.h>
main() {
int x, y;
int * const ptr = &x;

ptr = &y;
return 0;
}
```

```
Compiling FIG7_13.C:
Error FIG7__13.C 10: Cannot modify a const object
Waraing FI67_13.C 12:   lptr' is assigned a value that is never used
Warning FIG7_13.C 12:   'y'  is declared but never used
```

Fig. 7.13 Tentando modificar um ponteiro constante para dados nao-constantes.

O privilégio de acesso mínimo é garantido por um ponteiro constante para um dado constante. Tal ponteiro sempre aponta para o mesmo local da memória, e os dados nesse local da memória não podem ser modificados. E assim que um array deve ser passado a uma função que apenas verifica o array usando a notação de subscritos e não modifica o array. O programa da Fig. 7.14 declara uma variável ponteiro **ptr** do tipo **const int * const.** Essa declaração é lida da direita para a esquerda como **"ptr** é um ponteiro constante para uma constante inteira". A figura mostra as mensagens de erro geradas quando é feita uma tentativa de modificar os dados para os quais **ptr** aponta e quando é feita uma tentativa de modificar o endereço armazenado na variável ponteiro.

7.6 Classificação de Bolhas Usando Chamada por Referência

Vamos modificar o programa de classificação de bolhas da Fig. 6.15 de forma que ele utilize duas funções - **classBolha** e **swap.** A função **classBolha** realiza a ordenação do array. Ela chama a função **swap** para permutar os elementos do **array [j] earray[j+1]** do array (veja a Fig. 7.15). Lembre-se de que o C impõe a ocultação de informações entre funções, portanto **swap** não tem acesso a elementos individuais do array em **classBolha.** Como **classBolha** *deseja* que **swap** tenha acesso aos elementos do array a serem permutados, **classBolha** passa cada um desses elementos a **swap** por meio de uma chamada por referência — o endereço de cada elemento do array é passado explicitamente. Embora arrays inteiros sejam passados automaticamente por meio de chamadas por referência, os elementos individuais dos arrays são escalares e passados geralmente por meio de chamadas por valor. Portantos, **classBolha** usa o operador de endereços (&) em cada um dos elementos do array na chamada de **swap** da forma como se segue

swap(&array[j], &array[j + 1]);

para realizar a chamada por referência. A função **swap** recebe &**array[j**] na variável de ponteiro **elemento1Ptr.** Mesmo sem ter conhecimento do nome **array [j]** — devido à ocultação de infor mações —**, swap** pode usar **elemento1Ptr** como sinônimo de **array[j**]. Portanto, quando **swap** faz referência a ***elemento1Ptr,** na verdade está fazendo referência a **array [j**] em **classBolha.** Da mesma forma, quando **swap** faz referência a ***elemento2Ptr,** na realidade está fazendo **referência a um array [j + 1]** em **classBolha.** Muito embora **swap** não possa dizer

temp = array[j];
array[j] = array [j + 1];
array[j + 1] = temp;

exatamente o mesmo efeito é conseguido por

temp = *elemento1Ptr;
***elemento1Ptr = elemento2Ptr;**
***elemento2Ptr = temp;**

na função **swap** da Fig. 7.15.

```
/* Tentando modificar um ponteiro constante para */
/* dados constantes */
#include <stdio.h>

main() {
int x = 5, y;
const int *const ptr = &x;
*ptr = 7;
ptr = &y;
return 0;
}
```

```
Compiling FIG7-14.C:
Error FIG7-14.C 10: Cannot modify a const object
Error FIG7_14.C 11: Cannot modify a const object
Warning FIG7_14.C 13:   'ptr' is assigned a value that is never used
Warning FIG7__14.C 13:   'y' is declared but never used
```

Fig. 7.14 Tentando modificar um ponteiro constante para dados não-constantes.

```
/*Este programa coloca valores em um array,
classifica os valores em ordem ascendente e
imprime o array resultante */
#include <stdio.h>
#define TAMANHO 10

void classBolha  (int *, int);

main() {
int  i,   a[TAMANHO]= {2,6,4,8,10,12,89,68,45,37};

printf("Itens de dados na ordem original\n");
for(i = 0; i <= TAMANHO -  1; i++)
        printf("%4d", a[i]);

classBolha  (a, TAMANHO); /* ordena o array */
printf("\nltens de dados em ordem ascendente\n");
for   (i =  0;   i <= TAMANHO -  1; i++)
        printf("%4d", a[i]);
        printf("\n"); return 0;
}

void classBolha  (int *array,  int tamanho) {
int pass,  j;
void swap(int *,  int *);
for  (pass = 1; pass <= tamanho - 1; pass++)
        for  (j = 0;  j <= tamanho - 2; j++)
                if   (array[j]  > array[j + 1])
                        swap(iarray[j],  &array[j +1]);
}

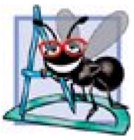
void swap(int *elementolPtr,  int *elemento2Ptr){
int temp;
temp = *elementolPtr;
*elementolPtr = *elemento2Ptr;
*elemento2Ptr = temp;
}
```

```
Itens de dados na ordem original
2  6   4  8   10     12     89     68     45 37
Itens de dados em ordem ascendente
2  4   6  8   10     12     37     45     68 89
```

Fig. 7.15 Classificação de bolhas com chamada por referência.

Vários aspectos da função **classBolha** devem ser observados. O cabeçalho da função declara **array** como int*arrayemvezdeint **array[]** para indicar que **classBolha** recebe um array unidimensional como argumento (mais uma vez, essas notações são intercambiáveis). O parâmetro **tamanho** é declarado **const para** impor o princípio do privilégio mínimo. Embora o parâmetro **tamanho** receba uma cópia do valor em **main** e modificar a cópia não modifique o valor em **main, classBolha** não precisa alterar **tamanho** para realizar sua tarefa. O tamanho do array permanece fixo durante a execução de **classBolha.** Portanto, **tamanho** é declarado **const** para assegurar que não será modificado. Se o tamanho do array fosse modificado durante o processo de ordenação, é possível que o algoritmo de ordenação não funcionasse corretamente.

O protótipo da função **swap** está incluído no corpo da função **classBolha** porque esta última é a única função que chama **swap.** Colocar o protótipo em **classBolha** restringe as chamadas de **swap àquelas** feitas a partir de **classBolha.** Outras funções que tentem chamar **swap** não terão acesso ao protótipo adequado da função e portanto o compilador gerará um automaticamente. Isso resulta normalmente em um protótipo que não é equivalente ao cabeçalho da função (e gera um erro de compilação) porque o compilador assume **int** para o tipo de retorno e os tipos dos parâmetros.

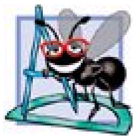


Observação de engenharia de software 7.4

Colocar protótipos de funções nas definições de outras funções impõe o princípio do privilégio mínimo restringindo as chamadas corretas às funções nas quais o protótipo aparece.

Observe que **classBolha** recebe o tamanho do array como parâmetro. A função deve saber o tamanho do array para ordená-lo. Quando um array é passado para uma função, o endereço do primeiro elemento na memória é recebido pela função. O endereço não fornece qualquer informação a respeito do número de elementos do array. Portanto, o programador deve fornecer à função o tamanho do array.

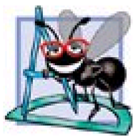
No programa, o tamanho do array foi passado explicitamente à função **classBolha.** Há duas vantagens principais desse método — a reutilização do software e a engenharia de software adequada. Definindo a função de modo que ela receba o tamanho do array como argumento, permitimos que ela seja usada por qualquer programa que classifique arrays inteiros unidimensionais, e os arrays podem ter qualquer tamanho.



Observação de engenharia de software 7.5

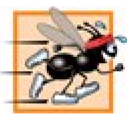
Ao passar um array para uma função. passe também o tamanho do array. Isso ajuda a generalizar a função. As funções generalizadas são reutilizadas freqüentemente em muitos programas.

Poderíamos ter armazenado o tamanho do array em uma variável global à qual todo o programa tivesse acesso. Isso seria mais eficiente porque não seria feita uma cópia do tamanho para ser passada à função. Entretanto, outros programas que exigissem a capacidade de classificar um array inteiro poderiam não ter a mesma variável global e portanto a função não poderia ser utilizada ali.



Observação de engenharia de software 7.6

As variáveis globais violam o princípio do privilégio mínimo e são um exemplo de engenharia de software deficiente.



Dica de desempenho 7.2

Passar o tamanho de um array para uma função toma tempo e exige um espaço de pilha adicional porque é feita uma cópia do tamanho para ser passada à função. Entretanto, as variáveis globais não exigem tempo ou espaço adicional porque podem ser acessadas diretamente por qualquer função.

O tamanho do array poderia ter sido programado diretamente na função. Isso restringiria o uso da função para um tamanho específico de array e reduziria tremendamente a reutilização daquela função. Apenas os programas que processassem arrays inteiros unidimensionais do tamanho específico codificado na função poderiam utilizá-la

A linguagem C fornece o *operador unário* especial *sizeof* para determinar o tamanho cm byte de um array (ou qualquer outro tipo de dado) durante a compilação de um programa. Ao ser explicado nome de um array como na Fig. 7.16, o operador sizeof retorna o número total de bytes no array como um inteiro. Observe que variáveis do tipo float são armazenadas normalmente em 4 bytes de memória, e array é declarado possuir 20 elementos. Portanto, há um total de 80 bytes no array.

O número de elementos em um array também pode ser determinado em tempo de compilação. Por exemplo, examine a seguinte declaração de array:

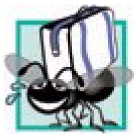
double real[22];

Normalmente, as variáveis do tipo double são armazenadas em 8 bytes de memória. Assim, o array real contém um total de 176 bytes. Para determinar o número de elementos do array, pode ser usada a expressão a seguir:

sizeof(real) / sizeof(double)

A expressão determina o número de bytes do array real e divide esse valor pelo número de bytes usado na memória para armazenar um valor double.

O programa da Fig. 7.17 calcula o número de bytes utilizado para armazenar cada um dos tipos padronizados de dados em um equipamento compatível com os PCs.



Dicas de portabilidade 7.2

O número de bytes utilizado para armazenar um determinado tipo de dado pode variar entre sistemas. Ao escrever programas que dependem dos tamanhos dos tipos de dados e que serão executados em compiladores de vários sistemas, use sizeof para determinar o número de bytes usado para armazenar os tipos de dados.

O operador sizeof pode ser aplicado a qualquer nome de variável, tipo ou constante. Ao ser aplicado a um nome de variável (que não seja um nome de array) ou uma constante, é retornado o número de bytes usado para armazenar o tipo específico de variável ou constante. Observe que os parênteses usados com **sizeof** são obrigatórios se o nome de um tipo for fornecido como seu operando. Omitir os parenteses resulta em um erro de sintaxe. Os parênteses não são exigidos se for fornecido um nome de váriavel como seu operando.

```
/* O operador sizeof ao ser utilizado em um nome de */
/* array retorna o numero de bytes do array */
#include <stdio.h>

main() {
float array [20];

printf(“O numero de bytes do array e %d\n”
          syzeof(array));
return 0;
}
```

```
O numeor de bytes do array e 80
```

Fig. 7.16 Ao ser aplicado a um nome de array, o operador sizeof retorna o número de bytes do array

```
/* Demonstrando o operador sizeof*/
#include <stdio.h>
main() {

printf(“sizeof(char) = %d”
        “sizeof(short) = %d”
         “sizeof(int) = %d”
        “sizeof(long) = %d”
        “sizeof(float) = %d”
        “sizeof(double) = %d”
        “sizeof(long double) = %d”,
```

```
sizeof(char), sizeof(short), sizeof(int),
sizeof(long), sizeof(float), sizeof(double),
sizeof(long double));
return 0;
}
```

```
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 2
sizeof(long) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 10
```

Fig. 7.17 Usando o operador **sizeof** para determinar os tamanhos dos tipos padronizados de dados.

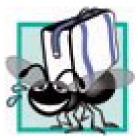
7.7 Expressões de Ponteiros e Aritmética de Ponteiros

Os ponteiros são operandos válidos em expressões aritméticas, expressões de atribuições e expressões de comparação. Entretanto, nem todos os operadores usados normalmente com essas expressões são válidos ao estarem junto de variáveis de ponteiros. Esta seção descreve os operadores que podem ter ponteiros como operandos e como estes operadores são utilizados.

Um conjunto limitado de operações aritméticas pode ser realizado com ponteiros. Um ponteiro pode ser incrementado (++) ou decrementado (--), pode ser adicionado um inteiro a um ponteiro (+ ou +=), um inteiro pode ser subtraído de um ponteiro (- ou - =) ou um ponteiro pode ser subtraído de outro. Admita que o array **int v[10]** foi declarado e seu primeiro elemento se encontra no local **3 0 0 0** na memória.

Admita que o ponteiro **vPtr** foi declarado e inicializado para apontar para **v [0**], i.e., o *valor* de **vPtr** é **3000**. A Fig. 7.18 mostra um diagrama dessa situação para um equipamento com inteiros de 4 bytes. Observe que **vPtr** pode ser inicializado para apontar para o array **v** com qualquer uma das seguintes instruções

```
vPtr = v;
vPtr = &v[0];
```



Dicas de portabilidade 7.3

A maioria dos computadores atuais contém inteiros de 2 ou 4 bytes. Alguns dos equipamentos mais recentes usam inteiros de 8 bytes. Em face de os resultados da aritmética de ponteiros dependerem do tamanho dos objetos para o qual um ponteiro aponta, tal aritmética é dependente do equipamento.

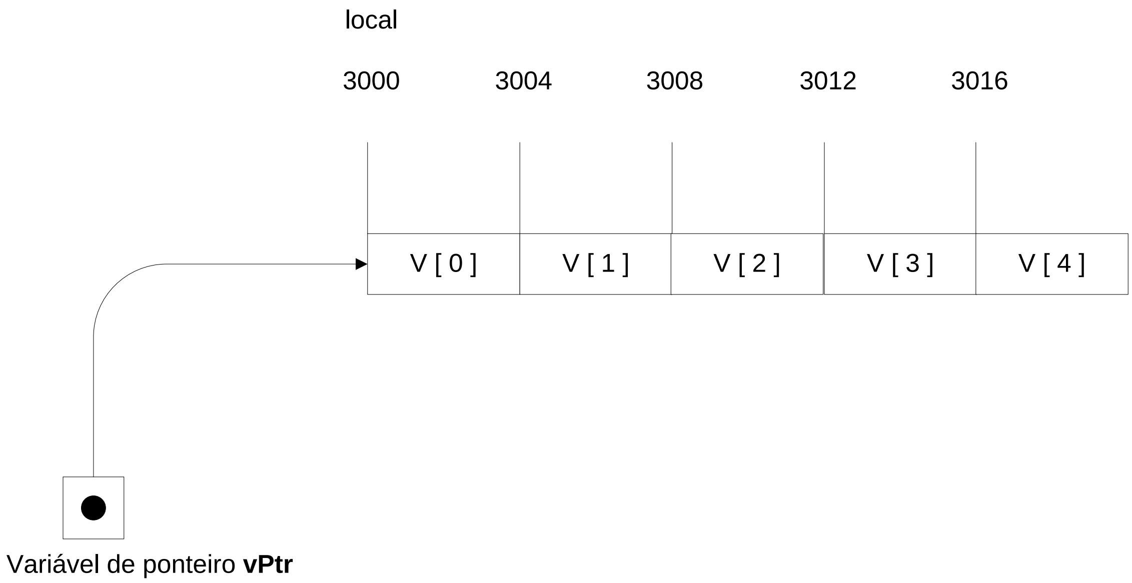


Fig. 7.18 O array **v** e uma variável de ponteiro **vPtr** que aponta para **v**.

Na aritmética convencional, a adição 3000 + 2 leva ao valor 3002. Normalmente, isso não acontece na aritmética de ponteiros. Quando um inteiro é adicionado ou subtraído de um ponteiro, este último não é simplesmente incrementado ou decrementado por tal inteiro, mas sim por tal inteiro vezes o tamanho do objeto ao qual o ponteiro se refere. O número de bytes depende do tipo de dado do objeto. Por exemplo, a instrução

vPtr += 2

produziria 3008 (3 000 + 2 * 4) admitindo que um inteiro está armazenado em 4 bytes de memória No array **v, vPtr** apontaria agora para **v [** 2] (Fig. 7.19). Se um inteiro for armazenado em 2 bytes de memória, o cálculo anterior resultaria na posição de memória 3004 (3000 + 2 * 2). Se o array fosse de um tipo diferente de dado, a instrução anterior incrementaria o ponteiro em duas vezes o numero de bytes necessário para armazenar um objeto desse tipo de dado. Ao realizar a aritmética de ponteiros em um array de caracteres, os resultados serão consistentes com a aritmética regular porque cada caractere tem o comprimento de um byte.

Se **vPtr** fosse incrementado para 3016, que aponta para **v** [4], a instrução

vPtr -= 4;

definiria novamente **vPtr** como 3000 — o início do array. Se um ponteiro for incrementado ou decrementado de um, os operadores de incremento e decremento podem ser usados. Qualquer uma das instruções

++vPtr;
vPtr++;

incrementa o ponteiro para apontar para a próxima posição no array. Qualquer uma das instruções

--vPtr;
vPtr--;

decrementa o ponteiro para apontar para a posição anterior do array.

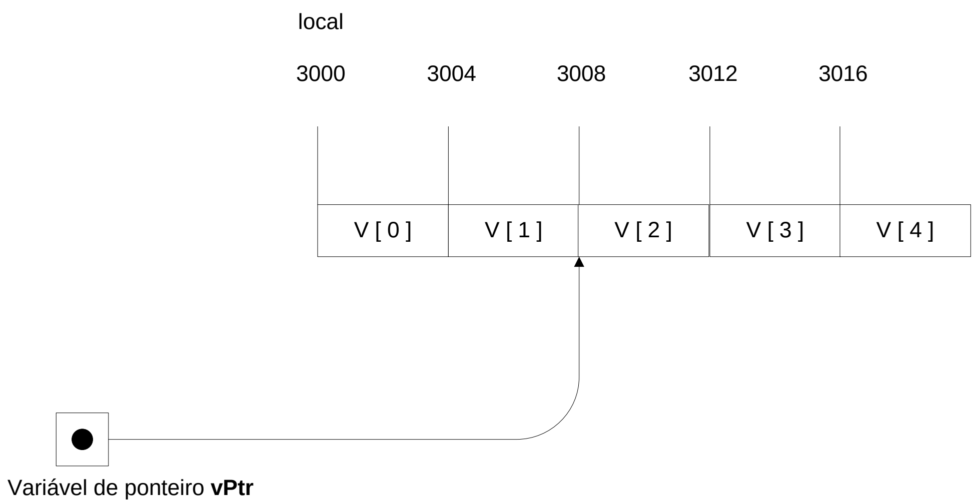
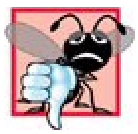


Fig. 7.19 O ponteiro **vPtr** após a aritmética de ponteiros.

As variáveis de ponteiros podem ser subtraídas entre si. Por exemplo, se **vPtr** possuir o local **3000 e v2Ptr** possuir o endereço **3008,** a instrução

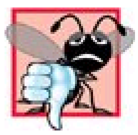
x = v2Ptr - vPtr;

atribuiria a **x** o número de elementos de array de **vPtr** a **v2Ptr,** nesse caso, **2.** A aritmética de ponteiros não tem significado algum se não for realizada em um array. Não podemos assumir que duas variáveis do mesmo tipo estejam armazenadas contiguamente na memória a menos que sejam elementos ad**jacentes** de um array.



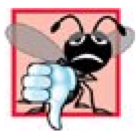
Erro comun de programação 7.5

Usar *a aritmética de ponteiros em um ponteiro que não se refere a um array de valores.*



Erro comun de programação 7.6

Subtrair ou comparar dois ponteiros que não se referem ao mesmo array.



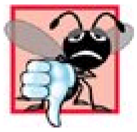
Erro comun de programação 7.7

Ultrapassar o final de um array ao utilizar a aritmética de ponteiros.

Um ponteiro pode ser atribuído a outro ponteiro se ambos forem do mesmo tipo. Caso contrário, deve ser utilizado um operador de conversão para transformar o ponteiro à direita da atribuição para o tipo do ponteiro à esquerda da mesma. A exceção a essa regra é o ponteiro para **void** (i.e., **void** *), que é um ponteiro genérico que pode representar qualquer tipo de ponteiro. Todos os tipos de ponteiros podem ser atribuídos

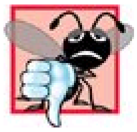
a um ponteiro a **void,** e um ponteiro a **void** pode ser atribuído a um ponteiro de qualquer tipo. Em ambos os casos, não é exigido um operador de conversão.

Um ponteiro a **void** não pode ser desreferenciado. Por exemplo, o compilador sabe que um ponteiro a **int** se refere a quatro bytes da memória de um equipamento com inteiros de 4 bytes, mas um ponteiro **a void** contém simplesmente um local da memória para um tipo desconhecido de dados — o número preciso de bytes para o qual o ponteiro se refere não é conhecido pelo compilador. O compilador deve conhecer o tipo de dado para determinar o número de bytes a ser desreferenciado para um ponteiro es pecífico. No caso de um ponteiro para **void,** esse número de bytes não pode ser determinado a partir do



Erro comun de programação 7.8

*Atribuir a um ponteiro de um tipo um ponteiro de outro tipo se nenhum deles for do tipo void * causa um erro de sintaxe.*



Erro comun de programação 7.9

*Desreferenciar um ponteiro void *.*

Os ponteiros podem ser comparados por meio de operadores de igualdade e relacionais, mas tais comparações não significam nada se os ponteiros não apontarem para membros do mesmo array. As comparações de ponteiros comparam os endereços armazenados nos ponteiros. Uma comparação de dois ponteiros apontando para o mesmo array poderia mostrar, por exemplo, que um ponteiro aponta para um elemento do array com maior numeração do que o elemento apontado pelo outro ponteiro. Um uso comum da comparação de ponteiros é para determinar se um ponteiro é NULL.

7.8 O Relacionamento entre Ponteiros e Arrays

Os arrays e os ponteiros estão intimamente relacionados em C e um ou outro podem ser usados quase indiferentemente. Pode-se imaginar que o nome de um array é um ponteiro constante. Os ponteiros podem ser usados para fazer qualquer operação que envolva um subscrito de array.



Dica de desempenho 7.3

A notação de subscritos de arrays é convertida para a notação de ponteiros durante a compilação, portanto escrever expressões de subscritos de arrays com a notação de ponteiros pode economizar tempo de compilação.



Boa prática de programação 7.5

Usar a notação de arrays em vez da notação de ponteiros ao manipular arrays. Embora o programa possa demorar um pouco mais para ser compilado, provavelmente ele ficará muito mais claro.

Admita que o array inteiro **b [5**] e a variável de ponteiro **bPtr** foram declarados. Como o nome do array (sem um subscrito) é um ponteiro para o primeiro elemento do array, podemos fazer com que bPtr seja igual ao endereço do primeiro elemento do array **b** com a instrução

bPtr = b;

Essa instrução é equivalente a tomar o endereço do primeiro elemento do array como se segue

bPtr = &b[0];

O elemento **b [3**] do array pode ser referenciado alternativamente com a expressão de ponteiro

***(bPtr + 3)**

O **3** na expressão anterior é o *offset (deslocamento)* do ponteiro. Quando um ponteiro aponta para o início de um array, o offset indica que elemento do array deve ser referenciado, e o valor do offset é idêntico ao subscrito do array. A notação anterior é chamada *notação ponteiro/offset.* Os parênteses são necessários porque a precedência do * é maior do que a precedência do +. Sem os parênteses, a expressão anterior adicionaria **3** ao valor da expressão ***bPtr** (i.e., seria adicionado **3** a **b [0**] admitindo que **bPtr** aponta para o início do array). Da mesma forma que o elemento do array pode ser referenciado por uma expressão de ponteiro, o endereço

&b[3]

pode ser escrito com a expressão de ponteiro

bPtr + 3

O array em si pode ser tratado como um ponteiro e usado com a aritmética de ponteiros. Por *exemplo,* a expressão

***(b + 3)**

também se refere ao elemento do array **b [3]**. Em geral, todas as expressões de arrays com subscritos podem ser escritas com um ponteiro e um offset. Nesse caso, a notação ponteiro/offset foi usada com o nome do array como ponteiro. Observe que a instrução anterior não modifica o nome do array de nenhuma forma; **b** ainda aponta para o primeiro elemento do array.

Os ponteiros podem possuir subscritos exatamente da mesma forma que os arrays. Por exemplo, a expressão

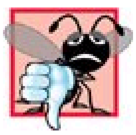
bPtr[1]

se refere ao elemento **b** [1] do array. Isso é chamado *notação ponteiro/subscrito.*

Lembre-se de que o nome de um array é basicamente um ponteiro constante; ele sempre aponta para o início do array. Dessa forma, a expressão

b += 3

é inválida porque tenta modificar o valor do nome do array com a aritmética de ponteiros.



Erro comun de programação 7.10

Tentar modificar o nome de um array com a aritmética de ponteiros é um erro de sintaxe.

O programa da Fig. 7.20 usa os quatro métodos que analisamos para fazer referência a elementos de -um array — subscritos de arrays, ponteiro/offset com o nome do array como ponteiro, subscrito de ponteiros ponteiro/offset com um ponteiro — para imprimir os quatro elementos do array inteiro **b**.

Para ilustrar ainda mais o intercâmbio entre arrays e ponteiros, vamos examinar as duas funções de cópia de strings — **copy1** e **copy2** — do programa da Fig. 7.21. Ambas as funções copiam uma string (provavelmente um array de caracteres) em um array de caracteres. Depois de uma comparação entre os protótipos das funções copy1 e **copy2,** elas parecem idênticas. Elas realizam a mesma tarefa; entretanto, são implementadas de maneira diferente.

A função copy1 usa a notação de subscritos de array para copiar a string em **s2** para a string em s1. A função declara uma variável inteira i como contador para ser usada como subscrito de array. O cabeçalho da estrutura for realiza toda a operação de cópia — seu corpo é uma instrução vazia. O cabeçalho especifica que i é inicializada com o valor zero e incrementada do valor um em cada iteração do loop. A condição na estrutura for, s1[i] = **s2**[i], realiza a operação de cópia de um caractere por ***vez.*** de **s2** para s1. Quando o caractere null for encontrado em s**2,** ele é atribuído a s1, e o loop

termina porque o valor inteiro do caractere null é zero (falso). Lembre-se de que o valor de uma instrução de atribuição é o valor atribuído ao argumento da esquerda.

A função **copy2** usa ponteiros e a aritmética de ponteiros para copiar a string em **s2** para o array de caracteres s1. Mais uma vez, o cabeçalho da estrutura for realiza toda a operação de cópia. O cabeça lho não inclui qualquer variável de inicialização. Como na função **copy1,** a condição **(*s1 = *s2)** realiza a operação de cópia. O ponteiro **s2** é desreferenciado e o caractere resultante é atribuído ao ponteiro desreferenciado **s 1.** Depois da atribuição na condição, os ponteiros são incrementados de forma a apontar para o próximo elemento do array **s1** e o próximo caractere da string **s2,** respectivamente. Quando o caractere null for encontrado em **s2,** ele é atribuído ao ponteiro desreferenciado **s1** e o loop se encerra.

```
/* Usando as notações de subscrito e ponteiro com arrays */
#include <stdio.h>

main() {
int i,  offset, b[]  =  {10,  20,  30, 40};
int *bPtr = b;   /* define bPtr para apontar para o array b */

printf(  "Array b impresso com:\n"
         "Notação de subscrito de array\n");

for  (i = 0;   i <= 3; i++)
         printf("b[%d]   = %d\n",  i, b[i]);
printf("\nNotacao ponteiro/offset em gue \n" "o ponteiro e o nome do array\n");
for (offset = 0; offset <= 3; offset++)
         printf("*(b + %d)  = %d\n",  offset,  *(b + offset));

printf("\nNotacao de subscrito de ponteiro\n");
for  (i = 0;   i <= 3; i++)
         printf("bPtr[%d]  = %d\n",  i, bPtr[i]);

printf("XnNotacao ponteiro/offset\n");

for  (offset = 0; offset <= 3; offset++)
         printf("*(bPtr + %d)  = %d\n", offset,  *(bPtr + offset));

return 0;
}
```

```
Array b impresso com:
Notação de subscrito de array
b[0]  = 10
b[l] = 20
b[2]  = 30
b[3]  = 40

Notação ponteiro/offset em que
o ponteiro e o nome do array
```

```
*(b + 0)  = 10
*(b + 1)  = 20
*(b + 2)  = 30
*(b + 3)  = 40

Notação de subscrito de ponteiro
bPtr[0]  = 10
bPtr[l]  = 20
bPtr[2]  = 30
bPtr[3]  = 40

Notação ponteiro/offset
*(bPtr +0) =10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr+ 3) = 40
```

Fig. 7.20 Usando quatro métodos para fazer referência aos elementos de um array

Observe que o primeiro argumento tanto de **copy1** como de **copy2** deve ser um array grande o suficiente para conter a string do segundo argumento. Caso contrário, pode ocorrer um erro quando for feita uma tentativa para gravar em um local da memória que é parte do array. Observe também que o segundo parâmetro de cada função é declarado como **const char** * (uma string constante). Em ambas as funções, o segundo argumento é copiado no primeiro argumento — é lido um caractere por vez, mas os caracteres nunca são modificados. Portanto, o segundo parâmetro é declarado de forma a apontar para um valor constante de forma que o princípio do privilégio mínimo seja imposto. Nenhuma das funções precisa ter a capacidade de modificar o segundo argumento, portanto essa capacidade não é fornecida a nenhuma delas.

7.9 Arrays de Ponteiros

Os arrays podem conter ponteiros. Um uso comum de tais estruturas de dados é para formar um array *de strings*.. Cada elemento do array é uma string. mas em C uma string é essencialmente um ponteiro para seu primeiro caractere. Assim, cada elemento de um array de strings é na verdade ura ponteiro para o primeiro caractere de uma string. Veja a declaração do array de strings **naipe** que pode ser útil para representar um baralho.

char *naipe[4] = {"Copas", "Ouros", "Paus", "Espadas"};

A parte **naipe [4**] da declaração indica um array de quatro elementos. A parte **char** * da declaração indica que cada elemento do array **naipe** é do tipo "ponteiro a **char". Os** quatro valores a serem colocados no array são "**Copas", "Ouros", "Paus", "Espadas**". Cada um deles é armazenado na memória como uma string de caracteres terminada em NULL que tem o comprimento de um caractere a mais do que o número de caracteres entre aspas. As quatro strings possuem comprimento de 6, 6, 5 e 8 caracteres, respectivamente. Embora pareça que essas strings estão sendo colocadas no array **naipe,** apenas os ponteiros são de fato colocados no array (Fig. 7.22). Cada ponteiro aponta para o primeiro caractere de sua string correspondente. Dessa forma, mesmo que o array **naipe** tenha seu tamanho definido. o ponteiro fornece acesso a strings com qualquer quantidade de caracteres. Essa flexibilidade é um exemplo dos poderosos recursos de estruturação de dados.

```
/*Copiando uma string usando a notação de array
e a notação de ponteiro */
#include <stdio.h>

void copyl(char *, const char *);
void copy2(char *,  const char *);

main()
{
char stringl[10], *string2 = "Hello",
string3[10],  string4[]  =  "Good Bye";

copyl(stringl, string2);
printf("stringl = %s\n", stringl);

copy2(string3, string4);
printf("string3 = %s\n", string3);

return 0;
}

/* copia s2 para sl usando a notação de array */
void copyl(char *sl,  const char *s2)
{
int i;
```

```
for  (i = 0;   sl[i]   = s2[i]; i++)
;/* nada e feito no corpo do loop */
}
/* copia s2 para sl usando a notação de ponteiro */
void copy2(char *sl,  const char *s2){
for ( ; *sl = *s2; sl++, s2++)
; /* nada e feito no corpo do loop */
}
```

```
stringl = Hello
string3 = Good Bye
```

Fig. 7.21 Copiando uma string usando a notação de array e a notação de ponteiro,

Os naipes poderiam ser colocados em um array bidimensional no qual cada linha representaria um naipe e cada coluna representaria uma letra do nome de um naipe. Tal estrutura de dados precisaria ter um numero fixo de colunas por linha, e esse número precisaria ter comprimento igual ao da maior string. Portanto, uma quantidade considerável de memória poderia ser desperdiçada quando um grande número de strings fosse armazenado com muitas strings menores do que a maior delas. Na próxima seção, usamos arrays de strings para representar um baralho.

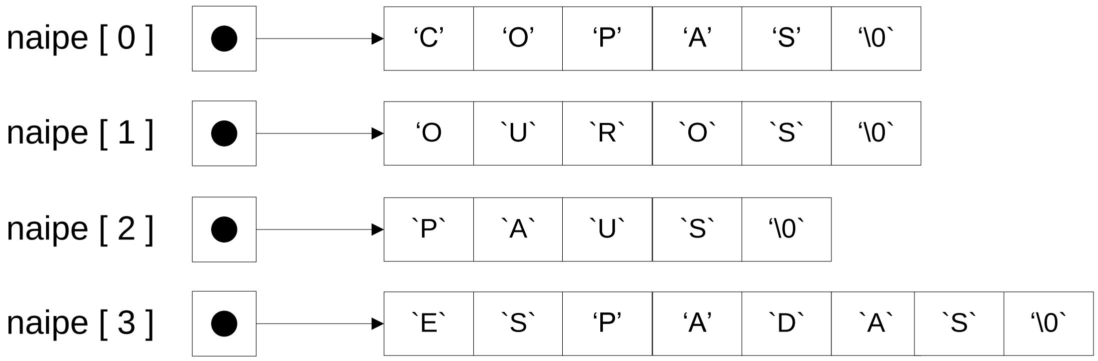


Fig. 7.22 Um exemplo gráfico do array **naipe.**

7.10 Estudo de Caso: Embaralhar e Distribuir Cartas

Nesta seção, usamos a geração de números aleatórios para desenvolver um programa de simulação de embaralhamento e distribuição de cartas. Esse programa pode ser usado para implementar programas que executam alguns jogos específicos de cartas. Para revelar alguns problemas sutis de perfomance, não usamos intencionalmente os melhores algoritmos para embaralhar e distribuir cartas. Nos exercicios e no Capítulo 10, desenvolvemos algoritmos mais eficientes.

Usando o método de refinamento top-down em etapas, desenvolvemos um programa que embalhará um baralho de 52 cartas e depois distribuirá cada uma delas. O método top-down é particularmente útil na solução de problemas maiores e mais complexos do que os que vimos nos capítulos anteriores.

Usamos um array bidimensional 4 por 13 **baralho** para representar o baralho de cartas (Fig. 7.23 As linhas correspondem aos naipes — a linha 0 corresponde ao naipe de copas, a linha 1 corresponde ao naipe de ouros, a linha 2. ao naipe de paus, e a linha 3, ao naipe de espadas. As colunas correspondem aos valores de face das cartas — as colunas 0 a 9 correspondem às cartas ás a dez, respectivamente, e as colunas 10 a 12 correspondem ao valete, à dama e ao rei. Devemos carregar o array **naipe com as** strings representando os quatro naipes e o array **face** com as strings de caracteres representando os treze valores das faces das cartas.

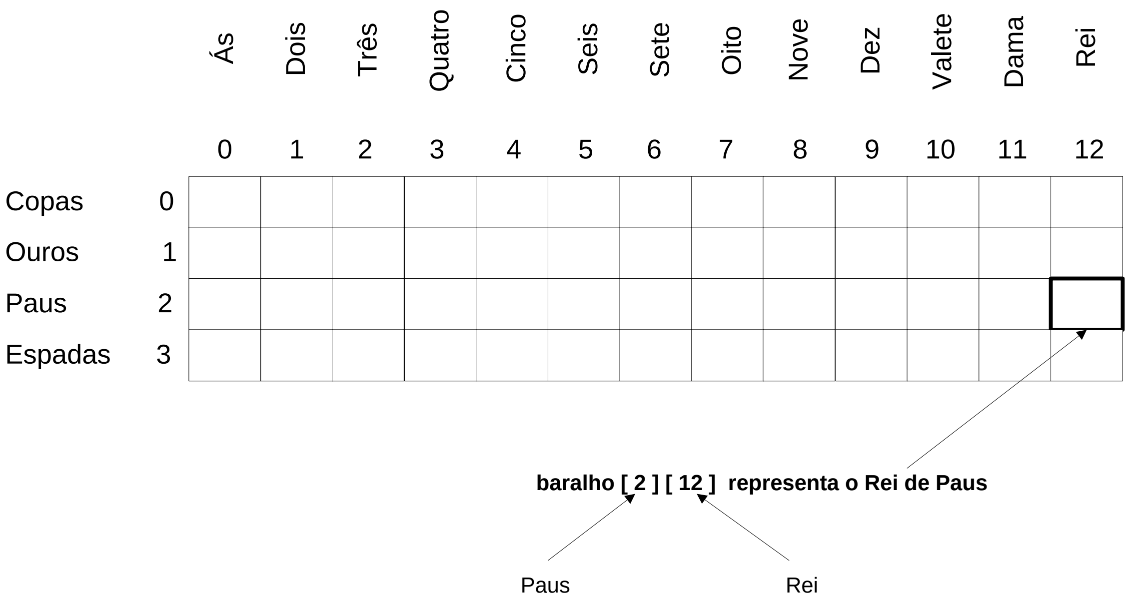


Fig. 7.23 Representação de um baralho como um array bidimencional.

Esse baralho simulado pode ser embaralhado como se segue. Em primeiro lugar os elementos do array **baralho** são igualados a zero. A seguir, uma **linha** (0-3) e uma **coluna** (0-12) são escolhidas aleatoriamente. O número 1 é inserido no elemento **baralho [linha] [coluna]** para indicar que essa carta vai ser a primeira do baralho embaralhado a ser distribuída. Esse processo continua com os numeros 2. 3.....52 sendo inseridos aleatoriamente no array **baralho** para indicar que cartas são colocadas em segundo, terceiro, ... e qüinquagésimo segundo lugar no baralho embaralhado. Quando o array baralho começar a ser preenchido com números de cartas, é possível que uma

carta seja selecionada duas vezes, i.e., baralho [linha] [coluna] não será zero quando a carta for selecionada. Essa seleção *é sim*plesmente ignorada e outras linhas e colunas são escolhidas repetidamente de maneira aleatória até que uma carta ainda não-selecionada seja encontrada. Mais tarde, os números 1 a 52 ocuparão os 52 espaços do array baralho. Nesse instante, o baralho está completamente embaralhado.

Esse algoritmo de embaralhamento poderia ser executado indefinidamente se as cartas que já foram selecionadas fossem selecionadas repetidamente de maneira aleatória. Esse fenômeno é conhecido como *retardamento indefinido.* Nos exercícios analisaremos um algoritmo melhor de embaralhamento que elimina a possibilidade de um retardamento indefinido.



Dica de desempenho 7.4

Algumas vezes um algoritmo que surge de uma "maneira " natural pode conter problemas de desempenho dificeis de detectar, como o retardamento indefinido. Procure utilizar algoritmos que evitem o retardamento indefinido.

Para distribuir a primeira carta, procuramos baralho [linha] [coluna] = 1 no array. Isso é realizado com uma estrutura for aninhada que varia linha de 0 a 3 e coluna de 0 a 12. A que carta o espaço do array corresponde? O array baralho já foi carregado com quatro naipes, e assim, para obter o naipe, imprimimos a string de caracteres naipe [linha]. Da mesma forma, para obter o valor de face da carta, imprimimos a string de caracteres face [coluna]. Também imprimimos a string de caracteres " de ". Imprimir essas informações na ordem adequada nos permite imprimir cada carta na fora "Rei de Paus", "As de Ouros" e assim por diante.

Vamos realizar o processo de refinamento top-down por etapas. O topo (top) é simplesmente

Embaralhar e distribuir 52 cartas

Nosso primeiro refinamento leva a:

Inicializar o array dos naipes
Inicializar o array do valor das faces das cartas
Inicializar o array do baralho
Embaralhar o baralho
Distribuir 52 cartas

"Embaralhar o baralho" pode ser expandido como se segue:

Para cada uma das 52 cartas
Colocar o número da carta em um espaço desocupado do baralho, selecionado aleatoriamente.

"Distribuir 52 cartas" pode ser expandido como se segue:

Para cada uma das 52 cartas
Encontrar o número da carta no baralho e imprimir o valor da face e o naipe da carta

Incorporando essas expansões, nosso segundo refinamento completo fica:

Inicializar o array dos naipes
Inicializar o array do valor das faces das cartas
Inicializar o array do baralho
Para cada uma das 52 cartas
Colocar o número da carta em um espaço desocupado do baralho, selecionado aleatoriamente
Para cada uma das 52 cartas
Encontrar o número da carta no baralho e imprimir o valor da face e o naipe da carta

"Colocar o número da carta em um espaço desocupado do baralho, selecionado aleatoriamente" pode ser expandido como se segue:

Escolher aleatoriamente um espaço no baralho
Ao escolher um espaço do baralho já escolhido anteriormente
Escolher aleatoriamente um espaço no baralho Colocar o número da carta no espaço escolhido do baralho

"Encontrar o número da carta no baralho e imprimir o valor da face e o naipe da carta" pode ser expandido como se segue:

Para cada espaço no array do baralho Se o espaço tiver um número de carta
Imprimir o valor da face e o naipe da carta

Incorporando essas expansões, o terceiro refinamento fica:

Inicializar o array dos naipes
Inicializar o array do valor das faces das cartas
Inicializar o array do baralho
Para cada uma das 52 cartas
Escolher aleatoriamente um espaço no baralho
Ao escolher um espaço do baralho já escolhido anteriormente
Escolher aleatoriamente um espaço no baralho Colocar o número da carta no espaço escolhido do baralho
Para cada uma das 52 cartas
Para cada espaço no array do baralho
Se o espaço tiver o número de carta desejado Imprimir o valor da face e o naipe da carta

Isso completa o processo de refinamento. Observe que esse programa seria mais eficiente se as partes de embaralhamento e distribuição do algoritmo fossem combinadas de modo que cada carta fosse distribuída ao ser colocada no baralho. Decidimos programar essas operações separadamente porque em geral as cartas são distribuídas após serem embaralhadas (e não enquanto são embaralhadas).

O programa de embaralhamento e distribuição de cartas é mostrado na Fig. 7.24, e um exemplo de execução é mostrado na Fig. 7.25. Observe o uso do especificador de conversão ***%s*** para imprimir strings de caracteres nas chamadas a **printf.** O argumento correspondente na chamada de **printf deve ser** um ponteiro para **char** (ou para um array

char). Na função **distribuir,** a especificação de de forma-to **"%6s de %-7s"** imprime uma string de caracteres alinhada à direita em um campo de cinco caracteres seguida de" **de "** e de uma string de caracteres alinhada à esquerda em um campo de oito caracteres O sinal de menos em **%-8s** significa que a string está alinhada à esquerda em um campo com comprimento 8.

Há um ponto fraco no programa de distribuição de cartas. Depois de uma carta ser encontrada, mesmo que tenha sido encontrada na primeira tentativa, as duas estruturas **for** internas continuam a procurar nos elementos restantes de **baralho.** Nos exercícios e em um estudo de caso no Capítulo 10, corrigimos essa deficiência.

```
/* Programa de distribuição de cartas */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void embaralhar(int [][13]);
void distribuir(const int  [][13],  const char *[],  const char *[]);

main()
{
char *naipe[4]   =  {"Copas","Ouros","Paus","Espadas"}
char *face[13]  = {"As","Dois","Tres", "Quatro",
                      "Cinco","Seis","Sete","Oito",
                      "Nove","Dez","Valete","Dama", "Rei"};

int baralho[4][13]  = {0};
srand(time(NULL));
embaralhar(baralho);
distribuir(baralho,   face, naipe);
return 0;
}

void embaralhar(int wBaralho[] [13 ] ) {
int carta,  linha, coluna;
for (carta = 1; carta <= 52;  carta++) {
        linha = rand() % 4;
        coluna = randO % 13;
        while  (wBaralho[linha][coluna] != 0) {
                linha = randO % 4;
                coluna = rand() % 13;
        }
        wBaralho[linha][coluna]  = carta;
}
}

void distribuir(const int wBaralho[] [13],  const char *wFace[], const char *wNaipe[]){

int carta,  linha, coluna;
for (carta = 1; carta <= 52; carta++)
        for (linha = 0;  linha <= 3; linha++)
```

```
                    for  (coluna = 0; coluna <= 12; coluna++)
                         if  (wBaralho[linha][coluna]   == carta)
                              printf("%6s de %-7s%c",
                                   wFace[coluna], wNaipe[linha],
                                   carta % 2 == 0 ? '\n' :'\t');
}
```

Fig. 7.24 Programa de distribuição de cartas.

Dois de Copas	Seis de Paus
Dama de Copas	Sete de Ouros
As de Paus	As de Espadas
Rei de Espadas	As de Ouros
Tres de Paus	As de Copas
Rei de Copas	Dama de Ouros
Nove de Paus	Dama de Pau
Nove de Espadas	Sete de Copas
Quatro de Copas	Dez de Copas
Dama de Espadas	Dois de Paus
Oito de Ouros	Dez de Espadas
Nove de Ouros	Tres de Espadas
Valete de Ouros	Dez de Ouros
Sete de Paus	Quatro de Espadas
Cinco de Copas	Quatro de Ouros
Cinco de Ouros	Dez de Paus
Quatro de Paus	Seis de Ouros
Valete de Copas	Seis de Espadas
Valete de Paus	Oito de Copas
Sete de Espadas	Tres de Ouros
Oito de Espadas	Nove de Copas
Cinco de Espadas	Tres de Copas
Rei de Paus	Dois de Espadas
Rei de Ouros	Seis de Copas
Valete de Espadas	Cinco de Paus
Dois de Ouros	Oito de Paus

Fig. 7.25 Exemplo de execução do programa de distribuição de cartas.

7.11 Ponteiros para Funções

Um ponteiro para uma função contém o endereço da função na memória. No Capítulo 6, vimos que o nome de um array é realmente o endereço do primeiro elemento do array na memória. Similarmente, o nome de uma função é na realidade o endereço inicial, na memória, do código que realiza a tarefa da função. Os ponteiros para funções podem ser passados a funções, retornados de funções, armazenamos em arrays e atribuídos a outros ponteiros de funções.

Para ilustrar o uso de ponteiros para funções, modificamos o programa de classificação de bolhas da Fig. 7.15 para formar o programa da Fig. 7.26. Nosso novo programa consiste em **main** e nas funções **bolha, swap, ascendente** e **descendente.** A função **classBolha** recebe um ponteiro para uma função — tanto a função **ascendente** como a função **descendente** — como argumento além de um array inteiro e o tamanho do array. O programa pede ao usuário que escolha se o array deve ser colocado na ordem ascendente ou descendente. Se o usuário entrar com 1, é passado, para a função bolha um ponteiro para a função **ascendente,** fazendo com que o array seja colocado na ordem crescente Se o usuário entrar com o valor 2, é passado, para a função **bolha,** um ponteiro para a função des**cendente,** fazendo com que o array seja colocado na ordem **decrescente.** A saída do programa é mostrada na Fig. 7.27.

O parâmetro a seguir aparece no cabeçalho da função **bolha:**

int (*compare)(int, int)

Isso diz a **bolha** para esperar um parâmetro que é um ponteiro para uma função que recebe dois parâmetros inteiros e retorna um resultado inteiro. Os parênteses em torno de * **compare** são necessários porque * tem uma precedência menor do que os parênteses que envolvem os parâmetros da função. Se não tivéssemos incluído os parênteses, a declaração seria

int *compare(int, int)

que declara uma função que recebe dois inteiros como parâmetros e retorna um ponteiro para um inteiro.

O parâmetro correspondente no protótipo da função **bolha** é **int (*)(int, int)** Observe que apenas os tipos foram incluídos, mas para efeito de documentação o programador pode incluir nomes que o compilador ignorará.

A função passada a **bolha** é chamada em uma instrução **if** como se segue

if ((*compare)(work[count], work[count + 1]))

Da mesma forma que um ponteiro para uma variável é desreferenciado para permitir o acesso ao valor da variável, um ponteiro para uma função é desreferenciado para que a função seja utilizada.

```
/* Programa geral de classificação usando ponteiros de funções */
#include <stdlo.h>

#define TAMANHO 10 *

void bolha(int *,  const int,  int  (*)(int, int));
int ascendente(const int,  const int);
int descendente(const int,  const int);

main(){
int a[TAMANHO] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
int contador, ordem;

printf("Digite 1 para colocar na ordem ascendente, \n");
printf("Digite 2 para colocar na ordem descendente: ");
scanf ("&d",  &ordem);

printf("\nltens de dados na ordem original\n");
for  (contador)  = 0; contador <= TAMANHO - 1; contador++)
        printf("%4d", a[contador]);

if (ordem ==1) {
        bolha(a,  TAMANHO, ascendente);
        printf("\nltens de dados na ordem ascendente\n");
}
else {
        bolha(a,  TAMANHO, descendente);
        printf("\nltens de dados na ordem descendente\n");
}
        for  (contador = 0; contador <= TAMANHO - 1; contador++)
                printf("%4d",  a[contador]);

        printf("\n");

return 0;
}

void bolha(int *work,  const int tamanho,  int  (*compare)(int, int))
{
int pass, count;
void swap(int *,  int *);
for  (pass = 1; pass <= tamanho - 1; pass++)
        for(count = 0;  count <= tamanho - 2; count++)
                if  ((*compare)(work[count], work[count + 1]))
                        swap(&work[count],  &work[count + 1]);
}

void swap(int *elementolPtr, int *elemento2Ptr) {
int temp;
temp = *elementolPtr;
```

```
*elementolPtr = *elemento2Ptr;
*elemento2Ptr = temp;
}

int ascendente(const int a, const int b) {
        return b < a;
}

int descendente(const int a,  const int b){
        return b > a;
}
```

Fig. 7.26 Programa geral de classificação usando ponteiros de funções

A chamada à função poderia ter sido feita sem desreferenciar o ponteiro como em

if (compare(work[count], work[count +1]))

que usa o ponteiro diretamente como o nome da função. Preferimos o primeiro método de chamar uma função por meio de um ponteiro porque esse método ilustra claramente que **compare** é um ponteiro para uma função que é desreferenciada para chamar uma função. O segundo método de chamar uma função por meio de um ponteiro faz parecer que **compare** é uma função real. Isso pode parecer confuso para um usuário do programa que gostasse de ver a definição da função **compare** e descobrisse que ela não está definida no arquivo.

Um uso comum de ponteiros de funções acontece nos conhecidos sistemas baseados em menus (menu driven systems). É solicitado a um usuário que seja selecionada uma opção de um menu (possívelmente de 1 a 5). Cada opção é atendida por uma função diferente. Os ponteiros para cada função são armazenados em um array, e o ponteiro no array é usado para chamar a função.

```
Digite 1 para colocar na ordem ascendente.
Digite 2 para colocar na ordem descendente: 1
Itens de dados na ordem original
2    6    4    8    10   12   89   68   45   37
Itens de dados na ordem ascendente
2    4    6    8    10   12   37   45   68   89
```

```
Digite 1 para colocar na ordem ascendente.
Digite 2 para colocar na ordem descendente: 2
Itens de dados na ordem original
2    6    4    8    10   12   89   68   45   37
Itens de dados na ordem descendente
89   68   45   37   12   10   8    6    4    2
```

Fig. 7.27 Saidas do programa de classificação de bolhas da Fig. 7.26.

O programa da Fig. 7.28 fornece um exemplo genérico do método de declarar e usar um array de ponteiros a funções. São definidas três funções — **funcao1, funcao2** e **funcao3** — que utilizam um argumento inteiro cada uma e nada retornam. Os ponteiros para essas três funções são armazenados *no* array **f** que é declarado como se segue:

void (*f[3])(int) = {funcao1, funcao2, funcao3};

```
/* Demonstrando um array de ponteiros a funções */
#include <stdio.h>

void funcaol(int);
void funcao2(int);
void funcao3(int);

main () {
void (*f[3])(int)  = {funcaol,  funcao2, funcao3};
int opcao;

printf("Digite um numero entre 0 e 2,  3 para finalizar: ");
scanf{"%d", &opcao);

while (opcao >= 0 && opcao < 3) {
        (*f[opcao])(opcao);
        printf("Digite um numero entre 0 e 2, 3 para finalizar: ");
        scanf("%d", &opcao);
}

printf("Voce digitou 3 para finalizar\n");
return 0;
}

void funcaol(int a){
printf("Voce digitou %d e funcaol foi chamada\n\n", a);
}

void funcao2(int b){
printf("Voce digitou %d e funcao2 foi chamada\n\n", b);
}

void funcao3(int c){
printf("Voce digitou %d e funcao3 foi chamada\n\n", c);
}
```

```
Digite um numero entre 0 e 2, 3 para finalizar: 0
Voce digitou 0 e funcaol foi chamada
Digite um numero entre 0 e 2, 3 para finalizar: 1
Voce digitou 1 e funcao2 foi chamada
Digite um numero entre 0 e 2, 3 para finalizar: 2
Voce digitou 2 e funcao3 foi chamada
Digite um numero entre 0 e 2, 3 para finalizar: 3
Voce digitou 3 para finalizar
```

Fig. 7.28 Demonstrando um array de ponteiros a funções

A declaração é lida a partir do conjunto de parênteses situado na extremidade esquerda, **"f** é um array de 3 ponteiros para funções que utilizam um **int** como argumento e retornam **void."** O array é inicializado com os nomes das três funções. Quando o usuário digitar um valor entre 0 e 2, o valor é usado como subscrito no array de ponteiros para funções. A chamada das funções é feita como se segue;

(*f[opcao])(opcao);

Na chamada, **f[opcao]** seleciona o ponteiro na posição **opcao** do array. O ponteiro é desreferenciados para chamar a função, e **opcao** é passado como argumento da função. Cada função imprime o o valor de seu argumento e o nome da função para indicar que a função foi chamada corretamente. Nos exercícios, você desenvolverá um sistema baseado em menus.

Resumo

• Ponteiros são variáveis que contêm endereços de outras variáveis como valores.
• Os ponteiros devem ser declarados antes de serem utilizados.
• A declaração

int *ptr;

declara **ptr** como ponteiro a um objeto do tipo **int** e é lida **"ptr** é um ponteiro para **int.** O ***** usado na declaração indica que a variável é um ponteiro.
• Há três valores que podem ser usados para inicializar um ponteiro: **0, NULL** ou um endereço. Inicializar um ponteiro com **0** e inicializar o mesmo ponteiro com **NULL** são procedimentos idênticos.
• O único inteiro que pode ser atribuído a um ponteiro é 0.
• O operador & (de endereço) retorna o endereço de seu operando.
• O operando do operador de endereço deve ser uma variável; o operador de endereço não pode ser. aplicado a constantes, a expressões ou a variáveis declaradas com a classe de armazenamento **register.**
• O operador *, chamado operador de referência indireta ou de desreferenciamento, retorna o valor do objeto para o qual seu operando aponta na memória. Chama-se a isso desreferenciar um ponteiro.
• Ao chamar uma função com um argumento que a função chamadora deseja que a função chamada modifique, o endereço do argumento é passado. A função chamada usa então o operador de referencia indireta (*) para modificar o valor do argumento na função chamadora.
• Uma função que recebe um endereço como argumento deve incluir um ponteiro como seu parametro formal correspondente.
• Não é necessário incluir os nomes dos ponteiros nos protótipos de funções; só é necessário incluir tipos dos ponteiros. Os nomes dos ponteiros podem ser incluídos para efeito de documentação, mas
o compilador os ignora.
• O qualificador **const** permite ao programador informar ao compilador que o valor de uma determinada variável não deve ser modificado.
• Se for feita uma tentativa para modificar um valor declarado **const,** o compilador a detecta e emite um aviso ou uma mensagem de erro, dependendo do compilador em particular.
• Há quatro maneiras de passar um ponteiro a uma função: um ponteiro não-constante para um dado não-constante, um ponteiro constante para um dado não-constante, um ponteiro não-constante para
um dado constante e um ponteiro constante para um dado constante.
• Os arrays são passados automaticamente por referência porque o valor do nome do array é o endere-ço do array.
• Para passar um elemento isolado de um array por meio de uma chamada por referência, o endereço do elemento específico do array deve ser passado.
• A linguagem C fornece o operador unário especial **sizeof** para determinar o tamanho em bytes de um array (ou qualquer outro tipo de dado) durante a compilação do programa.
• Ao ser aplicado ao nome de um array, o operador **sizeof** retorna o número total de bytes no array como um inteiro.
• O operador **sizeof** pode ser aplicado a qualquer nome de variável, tipo ou constante.

• As operações aritméticas que podem ser realizadas em ponteiros são incrementar (++) um ponteiro decrementar (--) um ponteiro, somar (+ ou +=) um ponteiro e um inteiro, subtrair (- ou -=) um ponteiro e um inteiro, e subtrair um ponteiro de outro.
• Quando um inteiro é adicionado ou subtraído de um ponteiro, o ponteiro é incrementado ou decrementado do inteiro vezes o tamanho do objeto para onde o ponteiro aponta.
• As operações aritméticas com ponteiros só devem ser realizadas em partes contíguas da memória como em um array. Todos os elementos de um array são armazenados contiguamente na memória,
• Ao realizar operações aritméticas com ponteiros em um array de caracteres, os resultados são como a aritmética normal porque cada caractere é armazenado em apenas um byte da memória,
• Um ponteiro pode ser atribuído a outro se ambos os ponteiros forem do mesmo tipo. Caso contrário, deve, ser utilizada uma conversão. A exceção a isso é um ponteiro a **void** que é um tipo de ponteiro genérico que pode conter ponteiros de qualquer tipo. Os ponteiros de outros tipos podem ser atribuídos a ponteiros a **void** e os ponteiros a **void** podem ser atribuídos a ponteiros de outros tipos sem conversão.
• Um ponteiro a **void** não pode ser desreferenciado.
• Os ponteiros podem ser comparados por intermédio dos operadores de igualdade e relacionais. Normalmente as comparações de ponteiros só fazem sentido se os ponteiros indicarem membros do mesmo array.
• Os ponteiros podem conter subscritos exatamente da mesma forma que os nomes de arrays.
• Um nome de array sem um subscrito é um ponteiro para o primeiro elemento do array.
• Na notação ponteiro/offset, o offset é o mesmo que um subscrito de array.
• Todas as expressões de arrays com subscritos podem ser escritas com um ponteiro e um offset usando tanto o nome do array como um ponteiro quanto um ponteiro separado que aponte para o array.
• Um nome de array é um ponteiro constante que sempre aponta para o mesmo local da memória. Os nomes dos arrays não podem ser modificados como os ponteiros convencionais, E possível ter arrays de ponteiros.
• É possível ter ponteiros para funções.
• Um ponteiro para uma função é o endereço onde se localiza o código da função.
• Os ponteiros para funções podem ser passados a funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros.
• Um uso comum de ponteiros para funções é nos conhecidos sistemas baseados em menus.

__Terminologia__

alocação dinâmica de memória
aritmética de ponteiros
array de strings
arrays de ponteiros
arrays de strings
atribuição de ponteiros
chamada por referência
chamada por valor
chamada simulada por referência
comparação de ponteiros
const
decrementar um ponteiro
desreferenciar um ponteiro
expressão de ponteiros
fazer referência direta a uma variável
fazer referência indireta a uma variável
íncrementar um ponteiro
indexar ponteiros
inicializar ponteiros
lista encadeada
notação ponteiro/offset offset
operador de desreferenciamento (*)
operador de endereço (&)
operador de referência indireta (*)
ponteiro
ponteiro constante
ponteiro constante para dado constante
ponteiro constante para dado não-constante
ponteiro de caractere ponteiro de função
ponteiro não-constante para dado constante
ponteiro não-constante para dado não-constante
ponteiro **NULL**
ponteiro para uma função
ponteiro para **void (void *)**
princípio do privilégio mínimo
referência indireta
refinamento top-down por etapas
retardamento indefinido
sizeof
somando um ponteiro a um inteiro
subscritos de ponteiros
subtrair dois ponteiros
subtrair um inteiro de um ponteiro
tipos de ponteiros
void * (ponteiro para **void)**

Erros Comuns de Programação

7.1 O operador de referência indireta * não se aplica a todos os nomes de variáveis em uma declaração. Cada ponteiro deve ser declarado com o * colocado antes do nome.

7.2 Desreferenciar um ponteiro que não foi devidamente inicializado ou que não foi atribuído para apontar para um local específico da memória. Isso poderia causar um erro fatal de tempo de execução, ou poderia modificar acidentalmente dados importantes e permitir que o programa seja executado até o final fornecendo resultados incorretos.

7.3 Não desreferenciar um ponteiro quando necessário para obter o valor para o qual o ponteiro aponta.

7.4 Não saber que uma função está aguardando ponteiros como argumentos de uma chamada por referencia e passar argumentos de chamada por valor. Alguns compiladores tomam os valores admitindo que são ponteiros e os desreferenciam como tais. Em tempo de execução, são geradas violações de acesso à memória ou falhas de segmentação. Outros compiladores captam incompatibilidade de tipos entre os argumentos e parâmetros e geram mensagens de erro.

7.5 Usar a aritmética de ponteiros em um ponteiro que não se refere a um array de valores.

7.6 Subtrair ou comparar dois ponteiros que não se referem ao mesmo array.

7.7 Ultrapassar o final de um array ao utilizar a aritmética de ponteiros.

7.8 Atribuir a um ponteiro de um tipo um ponteiro de outro tipo se nenhum deles for do tipo void * causa um erro de sintaxe.

7.9 Desreferenciar um ponteiro void *.

7.10 Embora os nomes de arrays sejam ponteiros para o início do array e os ponteiros possam ser modificados expressões aritméticas, os nomes de arrays não podem ser modificados em expressões aritméticas.

Práticas Recomendáveis de Programação

7.1 Incluir as letras ptr em nomes de variáveis de ponteiros para tornar claro que essas variáveis são ponteiros e precisam ser manipuladas apropriadamente.

7.2 Inicializar ponteiros para evitar resultados inesperados.

7.3 Usar uma chamada por valor para passar argumentos a uma função, a menos que o local chamador explicite que a função chamada modifique o valor da variável do argumento no ambiente original. Isso é outro exemplo do princípio do privilégio mínimo. Algumas pessoas preferem a chamada por referência por razões de desempenho porque o custo de copiar valores é evitado.

7.4 Antes de usar uma função, verifique o seu protótipo para determinar se ela é capaz de modificar os valores passados a ela.

7.5 Usar a notação de arrays em vez da notação de ponteiros ao manipular arrays. Embora o programa possa demorar um pouco mais para ser compilado, provavelmente ele ficará muito mais claro.

Dicas de Performance

7.1 Passe objetos grandes tais como estruturas usando ponteiros para dados constantes para obter as vantagens de desempenho da chamada por referência e a segurança da chamada por valor.

7.2 Passar o tamanho de um array para uma função toma tempo e exige um espaço de pilha adicional feita uma cópia do tamanho para ser passada à função. Entretanto, as variáveis globais não exigem tempo ou espaço adicional porque podem ser acessadas diretamente por qualquer função.

7.3 A notação de subscritos de arrays é convertida para a notação de ponteiros durante a compilação, portanto escrever expressões de subscritos de arrays com a notação de ponteiros pode economizar tempo de compilação.

7.4 Algumas vezes um algoritmo que surge de uma maneira "natural" pode conter problemas de desempenho difíceis de detectar, como o retardamento indefinido. Procure utilizar algoritmos que evitem o retardamento indefinido.

Dicas de Portabilidade

7.1 Embora const seja bem definido no ANSI C, alguns sistemas não a impõem.

7.2 O número de bytes utilizado para armazenar um determinado tipo de dado pode variar entre sistemas. Ao escrever programas que dependem dos tamanhos dos tipos de dados e que serão executados em compiladores de vários sistemas, use sizeof para determinar o número de bytes usado para armazenar os tipos de dados.

7.3 A maioria dos computadores atuais contém inteiros de 2 ou 4 bytes. Alguns dos equipamentos mais recentes usam inteiros de 8 bytes. Em face de os resultados da aritmética de ponteiros depender do tamanho dos objetos para o qual um ponteiro aponta, tal aritmética é dependente do equipamento.

Observações de Engenharia de Software

7.1 O qualificador **const** pode ser usado para impor o princípio do privilégio mínimo. Usar o princípio do privilégio mínimo para desenvolver software diminui tremendamente o tempo de depuração e os efeitos colaterais indesejáveis e torna um programa mais fácil de modificar e manter.

7.2 Se um valor não se modifica (ou não deve ser modificado) no corpo de uma função à qual é passado, ele deve ser declarado **const** para evitar que seja modificado acidentalmente.

7.3 Apenas um valor pode ser alterado em uma função chamada por valor. Esse valor deve ser atribuído pelo valor de retorno da função. Para modificar vários valores em uma função, deve-se utilizar chamada por referência.

7.4 Colocar protótipos de funções nas definições de outras funções impõe o princípio do privilégio mínimo, restringindo as chamadas corretas às funções nas quais o protótipo aparece.

7.5 Ao passar um array para uma função, passe também o tamanho do array. Isso ajuda a generalizar a função. As funções generalizadas são reutilizadas freqüentemente em muitos programas.

7.6 As variáveis globais violam o princípio do privilégio mínimo e são um exemplo de engenharia de software pobre.

Exercícios de Revisão

7.1 Responda ao que se segue:
a) Um ponteiro é uma variável que contém o __________ de outra variável como valor.
b) Os três valores que podem ser usados para inicializar um ponteiro são __________, ______________ ou um ______________.
c) O único inteiro que pode ser atribuído a um ponteiro é ____________________.

7.2 Diga se cada uma das sentenças a seguir é verdadeira ou falsa. Se a resposta for falsa, explique por quê. a) O operador de endereço & só pode ser aplicado a constantes, a expressões ou a variáveis declaradas com a classe de armazenamento **register.**
b) Um ponteiro declarado como **void** pode ser desreferenciado.
c) Um ponteiro não pode ser atribuído a outro ponteiro de tipo diferente sem um operador de conversão.

7.3 Faça o que se pede em cada uma das sentenças a seguir. Considere que os números flutuantes de precisão simples estão armazenados em 4 bytes e que o endereço inicial do array se situa na posição 1002500 na memória. Cada parte do exercício deve usar os resultados das partes anteriores quando apropriado.
a) Declare um array do tipo **float** chamado **números** com 10 elementos e inicialize os elementos com os valores **0. 0,1.1, 2 .2,9 .9.** Suponha que a constante simbólica **TAMANHO** foi definida como **10.**
b) Declare um ponteiro **nPtr** que aponte para um objeto do tipo **float.**
c) Imprima os elementos do array **números** usando a notação de subscrito de array. Use uma estrutura **for** e considere que a variável inteira de controle **i** foi declarada. Imprima cada número com 1 posição de precisão à direita do ponto decimal.
d) Forneça duas instruções diferentes que atribuam o endereço inicial do array **números** à variável de ponteiro **nPtr.**
e) Imprima os elementos do array **n]úmeros** usando a notação ponteiro/offset com o ponteiro **nPtr.**
f) Imprima os elementos do array **números** usando a notação ponteiro/offset com o nome do array como ponteiro.
g) Imprima os elementos do array **números** por meio de subscritos do ponteiro **nPtr.**
h) Faça referência ao elemento 4 do array **números** usando a notação de subscrito de array, a notação ponteiro/offset com o nome do array como ponteiro, a notação de subscrito de ponteiro com **nPtr** e a notação ponteiro/offset com **nPtr.**
i) Admitindo que **nPtr** aponta para o início do array **números,** que endereço é referenciado por **nPtr + 8?** Que valor está armazenado nesse local?
j) Admitindo que **nPtr** aponta para **números [5]**, que endereço é referenciado por **nPtr -= 4**?Que valor está armazenado nesse local?

7.4 Para cada uma das sentenças seguintes, escreva uma instrução que realize a tarefa indicada. Admita que as variáveis de ponto flutuante **numero1** e **numero2** foram declaradas e que **numero1** foi inicializada com o valor 7.3.
a) Declare a variável **fPtr** como ponteiro para um objeto do tipo **float.**
b) Atribua o endereço da variável numero1 à variável de ponteiro fPtr.
c) Imprima o valor do objeto apontado por fPtr.
d) Atribua o valor do objeto apontado por fPtr à variável numero2.
e) Imprima o valor de numero2.
f) Imprima o endereço de numero1. Use o especificador de conversão %p.

g) Imprima o endereço armazenado em fPtr. Use o especificador de conversão %p. O valor impresso é igual ao endereço de numero?

7.5 Faça o que se pede.

a) Escreva o cabeçalho de uma função chamada exchange que utiliza dois ponteiros para os números de ponto flutuante x e y como parâmetros e não retorna um valor.

b) Escreva o protótipo da função do item (a).

c) Escreva o cabeçalho de uma função chamada evaluate que retoma um inteiro e utiliza como parâmetros um inteiro x e um ponteiro para uma função poly. A função poly utiliza um parâmetro inteiro e retoma um inteiro.

d) Escreva o protótipo da função do item (c).

7.6 Encontre o erro em cada um dos segmentos de programas a seguir. Admita que

int *zPtr; /* zPtr faz referencia ao array z */
int *aPtr = NULL;
void *sPtr = NULL;
int numero, i;
int z[5]   = {1,   2,   3,   4, 5);
sPtr = z

a) + + zPtr;

b) /* usa o ponteiro para obter o primeiro valor do array */ numero = zPtr;

c) /* atribui o elemento 2 do array (o valor 3) a numero */ numero = *zPtr[2];

d) /* imprime todo array z */ for (i = 0; i <= 5; i++)
printf("%d ", zPtr[i]);

e) /* atribui o valor apontado por sPtr a numero */ numero = *sPtr;

f) + + z;

Respostas dos Exercícios de Revisão

7.1 a) endereço, b) 0, NULL. um endereço, c) 0.

7.2 a) Falso. O operador de endereço só pode ser aplicado a variáveis, e não pode ser aplicado a variáveis com a classe de armazenamento register.
b) Falso. Um ponteiro para void não pode ser desreferenciado porque não há maneira de saber exatamente quantos bytes de memória devem ser desreferenciados.
c) Falso. Os ponteiros de outros tipos podem ser atribuídos a ponteiros do tipo void e ponteiros do tipo void podem ser atribuídos a ponteiros de outros tipos.

7.3 a) float números [TAMANHO] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
b) float *nPtr;
c) for (i = 0; i <= TAMANHO - 1; i++)
printf("%.1f ", numeros[i]);
d) nPtr = numeros;
nPtr = &numeros[0]; c) for (i = 0; i <= TAMANHO - 1; i++) printf("%.1f ", *(nPtr + i));
f) for (i = 0; i <= TAMANHO - 1; i + +)
printf("%.1f ", *(números + i));
g) for (i = 0; i <= TAMANHO - 1; i++)
printf ("9&.1f ", nPtr[i]);
h) números[4] *(números + 4) nPtr[4] *(nPtr +4)
i) O endereço é 1002500 + 8 * 4 = 1002532. O valor é 8 . 8. j) O endereço de números [5] é 1002500 + 5 * 4 = 1002520.
O endereço de nPtr -= 4 é 1002520 - 4 * 4 = 1002504.
O valor naquele local é 1.1. 7.1

7.4 a.) float *fPtr; b) fPtr = &numero1;
c) printf("O valor de *fPtr e %f\n", *fPtr);
d) numero2 = *fPtr;
e) printf("O valor de numero2 e %f\n", numero2);
f) printf("O valor de numero1 e %p\n", &numero1);
g) printf("O endereço armazenado em fptr e %p\n" fPtr);
Sim, o valor é o mesmo.

7.5
a) void exchange (float *x, float *y)
b) void exchange(float *, float *);
c) int evaluate(int x, int (*poly)(int)) d) int evaluate(int, int(*)(int));

7.6 a) Erro:zPtr não foi inicializado.
Correção: Inicialize zPtr com zPtr = z; b) Erro: O ponteiro não está desreferenciado.
Correção: Mude a instrução para numero = *zPtr;
c) Erro: zPtr [2] não é um ponteiro e não deve ser desreferenciado. Correção: Mude *zPtr [2] para zPtr [2].
d) Erro: Fazer referência a um elemento fora dos limites do array com subscritos de ponteiros. Correção: Mude o valor final da variável de controle na estrutura for para 4.
e) Erro: Desreferenciar um ponteiro void.
Correção: Para desreferenciar o ponteiro, primeiramente ele deve ser convertido em um ponteiro inteiro. Mude a instrução anterior para numero = *(int *)sPtr;
f) Erro: Tentar modificar um nome de array com a aritmética de ponteiros.

Correção: Use uma variável de ponteiro em vez de um nome de array para realizar a aritmética de ponteiros ou coloque subscrito no nome do array para fazer referência a um elemento específico.

Exercicios

7.7 Complete as seguintes sentenças:
a) O operador_retorna o local da memória onde o operando está armazenado.
b) O operador_retorna o valor do objeto para o qual seu operando aponta.
c) Para simular uma chamada por referência ao passar uma variável que não é um array a uma função, é necessário passar o(a)_da variável à função.

7.8 Diga se cada uma das sentenças seguintes é verdadeira ou falsa. Se falsa, explique por quê.
a) Dois ponteiros que apontam para arrays diferentes não podem ser comparados expressivamente.
b) Em face de o nome do array ser um ponteiro para o primeiro elemento do array, os nomes de arrays podem ser manipulados exatamente da mesma forma que os ponteiros.

7.9 Faça o que é pedido em cada uma das sentenças a seguir. Admita que inteiros sem sinal (unsigned) estão armazenados em 2 bytes e que o endereço inicial do array está no local 1002500 da memória.
a) Declare um array do tipo unsigned int chamado valores com 5 elementos e inicialize os elementos com inteiros pares de 2 a 10. Considere que a constante simbólica TAMANHO foi definida como 5.
b) Declare um ponteiro vPtr que aponte para um objeto do tipo unsigned int.
c) Imprima os elementos do array valores usando a notação de subscrito de array. Use uma estrutura for e admita que a variável de controle i foi declarada.
d) Forneça duas instruções diferentes que atribuam o endereço inicial do array valores à variável de ponteiro vPtr.
e) Imprima os elementos do array valores usando a notação ponteiro/offset.
f) Imprima os elementos do array valores usando a notação ponteiro/offset com o nome do array como ponteiro.
g) Imprima os elementos do array valores utilizando subscritos no ponteiro para o array.
h) Faça referência ao elemento 5 do array valores usando a notação de subscrito de array, a notação ponteiro/offset com o nome do array como ponteiro, a notação de subscrito de ponteiro e a notação ponteiro/offset.
i) Que endereço é referenciado por vPtr + 3? Que valor está armazenado nesse local?
j) Admitindo que vPtr aponta para valores [4], que endereço é referenciado por vPtr -= 4. Que valor está armazenado nesse local?

7.10 Para cada uma das sentenças a seguir, escreva uma única instrução que realize a tarefa indicada. Considere que as variáveis inteiras **valor1** e valor2, do tipo **long,** foram declaradas e que **valor1** foi inicializada com o valor 200000.
a) Declare a variável 1**Ptr** como ponteiro de um objeto do tipo **long.**
b) Atribua o endereço da variável **valor1** à variável de ponteiro 1**Ptr.**
c) Imprima o valor do objeto apontado por 1**Ptr.**
d) Atribua o valor do objeto apontado por 1**Ptr** à variável valor2.

e) Imprima o valor de valor2.
f) Imprima o endereço de **valor1.**
g) Imprima o endereço armazenado em 1**Ptr.** O valor impresso é igual ao endereço de **valor1?**

7.11 Faça o que se pede.
a) Escreva o cabeçalho da função **zero** que utiliza um parâmetro array inteiro **inteirosGrandes do** tipo **long** e não retorna um valor.
b) Escreva o protótipo da função do item **(a).**
c) Escreva o cabeçalho da função **somaUmValor** que utiliza um parâmetro array inteiro **valorPequeno** e retorna um inteiro.
d) Escreva o protótipo da função descrita na parte **(c).**

7.12 *Nota: Os Exercícios 7.12 a 7.15 são razoavelmente difíceis. Depois de ter resolvido esses problemas, voce deve ser capaz de implementar com facilidade os jogos de cartas mais conhecidos.*
7.12 Modifique o programa da Fig. 7.14 de forma que a função de distribuição de cartas distribua uma mão pôquer com cinco cartas. Depois escreva as seguintes funções adicionais:
a) Determine se a mão contem um par.
b) Determine se a mão contem dois pares.
c) Determine se a mão contem uma trinca (e.g., três valetes).
d) Determine se a mão contém uma quadra (e.g., quatro ases).
c) Determine se a mão contém um *flush* (e.g., todas as cinco cartas do mesmo naipe).
f) Determine se a mão contém uma seqüência (i.e.. cinco cartas com valores de face consecutivos). **J**

7.13 Use as funções desenvolvidas no Exercício 7.12 para escrever um programa que distribua duas mãos poquer com cinco cartas, avalie cada mão e determine a melhor.

7.14 Modifique o programa desenvolvido no Exercício 7.13 de forma que ele simule o distribuidor de cartas (a banca). A mão de cinco cartas do distribuidor é colocada com a "face para baixo" de forma que o jogador não a veja. O programa deve então avaliar a mão do distribuidor e, com base na qualidade das cartas, deve pedir
mais uma, duas ou três cartas para substituir o mesmo número de cartas desnecessárias na mão original. A seguir, o programa deve reavaliar a mão do distribuidor. *(Atenção:* Este é um problema difícil!)

7.15 Modifique o programa desenvolvido no Exercício 7.14 de forma que ele possa controlar automaticamente a mão de cartas do distribuidor, mas o jogador possa decidir que cartas de sua mão serão substituídas. O programa deve então avaliar ambas as mãos e determinar o vencedor. Agora use esse novo programa para fazer 20 jogos contra o computador. Quem venceu mais jogos, você ou o computador? Com base nesses resultados, faça as modificações apropriadas para refinar o programa de jogo de pôquer (este também é um problema difícil). Jogue mais 20 vezes. Seu programa modificado jogou melhor?

7.16 No programa de embaralhar e distribuir cartas da Fig. 7.24, usamos intencionalmente um algoritmo ineficiente de embaralhamento que apresenta a possibilidade de retardamento indefinido. Neste problema, você criará um algoritmo de embaralhamento de alto desempenho que evita o retardamento indefinido.

Modifique o programa da Fig. 7.24 como se segue. Comece inicializando o array **baralho** como mostra Fig. 7.29. Modifique a função **embaralhar** para fazer loops entre as linhas e as colunas do array, verificando um elemento por vez. Cada elemento deve ser permutado com um elemento do array selecionado aleatoriamente.

Imprima o array resultante para determinar se o baralho está embaralhado satisfatoriamente (como na Fig. 7.30, por exemplo). Você pode desejar que seu programa chame a função **embaralhar** várias vezes para assegurar um embaralhamento satisfatório.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	48	50	51	52

Fig. 7.29 Array **baralho** não-embaralhado.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Fig. 7.30 Exemplo de array **baralho** embaralhado.

Observe que, embora o método deste problema melhore o algoritmo de embaralhamento, o algoritmo de distribuição ainda exige a pesquisa no array **baralho** para procurar a carta 1, depois a carta 2, depois a 3 e assim sucessivamente. E o que é pior, mesmo depois de o algoritmo de distribuição localizar e distribuir a carta, a procura continua no restante do baralho. Modifique o programa da Fig. 7.24 de forma que, uma vez distribuída a carta, não sejam feitas mais tentativas de encontrar aquele número de carta e o programa passe imediatamente à distribuição da próxima carta. No Capítulo 10, desenvolvemos um algoritmo de distribuição que exige apenas uma operação por carta.

7.17 *(Simulação: A Lebre e a Tartaruga)* Neste problema você recriará um dos momentos verdadeiramente grandiosos da história, que é a clássica corrida entre a lebre e a tartaruga. Você usará a geração aleatória de números para desenvolver uma simulação desse memorável evento.

Nossos competidores começam a corrida no "quadrado 1" de setenta quadrados. Cada quadrado representa uma posição possível ao longo do trajeto da corrida. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar do quadrado 70 ganha uma cesta de cenouras e alfaces. O trajeto da corrida inclui uma subida pela encosta de uma montanha escorregadia, portanto, ocasionalmente, os competidores perdem terreno.

Há um relógio que emite um tique por segundo. A cada tique do relógio, seu programa deve ajustar a posição dos animais de acordo com as seguintes regras:

Animal	Tipo de movimento	Porcentagem do tempo	Movimento real
Tartaruga	Movimento rápido	50%	3 quadrados para a direita
	Escorregão	20%	6 quadrados para a esquerda
	Movimento lento	30%	1 quadrado para a direita
Lebre	Sono	20%	Absolutamente nenhum movimento
	Salto grande	20%	9 quadrados para a direita
	Grande escorregão	10%	12 quadrados para a esquerda
	Salto pequeno	30%	1 quadrado para a direita
	Pequeno escorregão	20%	2 quadrados para a esquerda

Animal	Tipo de movimento	Porcentagem do tempo	Movimento real
Tartaruga	Movimento rápido	50%	3 quadrados para a direita
	Escorregão	20%	6 quadrados para a esquerda
	Movimento lento	30%	1 quadrado para a direita
Lebre	Sono	20%	Nenhum movimento
	Salto grande	20%	9 quadrados para a direita
	Grande escorregão	10%	12 quadrados para esquerda
	Salto pequeno	30%	1 quadrado para a direita
	Pequeno escorregão	20%	2 quadrados para a esquerda

Use variáveis para controlar as posições dos animais (i.e., os números das posições vão de 1 a 70). Cada animal inicia na posição 1 (i.e., a "linha de partida"). Se um animal escorregar para antes do quadrado 1, leve o animal de volta para o quadrado 1.

Gere as porcentagens da tabela anterior produzindo um inteiro aleatório, i, no intervalo 1 < i < 10. Para a tartaruga, realize um "movimento rápido" quando 1 < i < 5, um "escorregão" quando 6 < i < 7 ou um "movimento lento" quando 8 <= i <= 10. Use uma técnica similar para mover a lebre.

Comece a corrida imprimindo

BANG !!!!!

E ELES PARTIRAM ! ! ! ! !

Depois, para cada tique do relógio (i.e., cada repetição do loop), imprima uma linha com 70 posições mostrando a letra **T** na posição da tartaruga e a letra **L** na posição da lebre. Ocasionalmente, os competidores estarão no mesmo quadrado. Nesse caso, a tartaruga morde a lebre e seu programa deve imprimir **OUCH**! I ! iniciando naquela posição. Todas as posições impressas diferentes das que levam o **T,** o **L** e a palavra **OUCH! ! !** (no caso de os animais ocuparem o mesmo quadrado) devem estar em branco.

Depois de cada linha ser impressa, teste se algum dos animais alcançou ou passou do quadrado 70. Em caso positivo, imprima o vencedor e termine a simulação. Se a tartaruga vencer, imprima **TARTARUGA**

VENCEU! ! ! YAY! ! ! Se a lebre vencer, imprima **Lebre venceu. Yuch.** Se ambos os animais vencerem no mesmo tique do relógio, você pode querer favorecer a tartaruga ("a parte mais fraca") ou pode desejar imprimir **Houve um empate.** Se nenhum animal vencer, realize o loop novamente para simular o próximo tique do relógio. Quando estiver pronto para executar seu programa, reúna um grupo de amigos para assistir à corrida. Você ficará surpreso com o entusiasmo da audiência!

Seção Especial: Construindo Seu Próprio Computador

Nos vários problemas que se seguem, nos desviamos temporariamente do mundo da programação em linguagem de alto nível. Vamos "remover a cobertura" de um computador e verificar sua estrutura interna. Apresentamos a programação em linguagem de máquina e escrevemos vários programas em linguagem de máquina. Para fazer com que isso seja uma experiência válida, construímos um computador (por intermédio da técnica de *simulação* baseada em software) no qual você pode executar seus programas em linguagem de máquina.

7.18 *(Programação em Linguagem de Máquina)* Vamos criar um computador a que chamaremos Simpletron. Como o nome já diz, ele é um equipamento simples, mas também, como veremos em breve, poderoso. O Simpletron executa programas escritos apenas na linguagem que entende diretamente, isto é, a Linguagem de Máquina Simpletron, ou, abreviadamente, LMS.

Código da operação	Significado
Operações de entrada/saída:	
#define READ 10	Lê uma palavra do terminal e a coloca em um local específico da memória.
#define WRITE 11	Escreve no terminal uma palavra de um local específico da memória.
Operações de carregamento/armazenamento:	
#define LOAD 20	Carrega no acumulador uma palavra de um local específico da memória.
#define STORE 21	Armazena em um local específico da memória uma palavra do acumulador.
Operações aritméticas:	
#define ADD 30	Adiciona uma palavra de um local específico da memória à palavra no acumulador (o resultado fica no acumulador).
#define SUBTRACT 31	Subtrai da palavra no acumulador, uma palavra em um local específico da memória (o resultado fica no acumulador).
#define DIVIDE 32	Divide uma palavra em um local específico da memória pela palavra no acumulador (o resultado fica no acumulador).
#define MULTIPLY 33	Multiplica uma palavra em um local específico da memória pela palavra no acumulador (o resultado fica no acumulador).
Operações de transferência de controle:	
#define BRANCH 40	Desvia para um local específico da memória.
#define BRANCHNEG 41	Desvia para um local específico da memória se o acumulador for negativo.
#define BRANCHZERO 42	Desvia para um local específico da memória se o acumulador for zero.
#define HALT 43	Término, i.e., o programa completou sua tarefa

Fig. 7.31 Códigos de operação da Linguagem de Máquina Simpletron (LMS).

O Simpletron contém um *acumulador* — um "registro especial" no qual as informações

são colocadas antes que o Simpletron as utilize em cálculos ou as examine de várias maneiras. Todas as informações no Simpletron são manipuladas em termos de *palavras.* Uma palavra é um número decimal de quatro dígitos e com sinal, como +3364, -1293, +0007, -0001 etc. O Simpletron está equipado com uma memória de 100 palavras, e faz-se referência a essas palavras por meio de seus números de localização 00,01, ...,99.

Antes de rodar um programa em LMS, devemos *carregar* ou colocar o programa na memória. A primeira instrução de qualquer programa em LMS é sempre colocada no local 00.

Cada instrução escrita em LMS ocupa uma palavra da memória do Simpletron (e assim as instruções são números decimais de quatro dígitos com sinal). Admitiremos que o sinal de uma instrução LMS é sempre positivo, mas o sinal de uma palavra de dados pode ser tanto positivo como negativo. Cada local da memória do Simpletron pode conter uma instrução, o valor de um dado usado por um programa ou uma área não-utilizada da memória (e portanto indefinida). Os dois primeiros dígitos de cada instrução LMS são o *código da operação,* que especifica a operação a ser realizada. Os códigos de operação da LMS estão resumidos na Fig. 7.31.
Os dois últimos dígitos de uma instrução LMS são o *operando,* que é o endereço do local da memória que contém a palavra à qual a operação se aplica. Agora vamos examinar vários programas simples em LMS.

Exemplo 1 Local	Número	Instrução
00	+1007	(Ler A)
01	+1008	(Ler B)
02	+2007	(Carregar A)
03	+3008	(Adicionar B)
04	+2109	(Armazenar C)
05	+1109	(Escrever C)
06	+4300	(Terminar)
07	+0000	(Variavel A)
08	+0000	(Variavel B)
09	+0000	(Resultado C)

Esse programa em LMS lê dois números do teclado, calcula sua soma e a imprime. A instrução +10 07 lê o primeiro número do teclado e o coloca no local 07 (que foi inicializado com o valor zero). A seguir, + 1008 lê o próximo número e o coloca no local 08. A instrução *carregar (load),* +2 007, coloca o primeiro número no acumulador, e a instrução *adicionar (add),* +3008, soma o segundo número ao número existente no acumulador. *Todas as instruções aritméticas LMS deixam seus resultados no acumulador.* A instrução *armazenar (store),* +2109, coloca o resultado novamente no local de memória 09 do qual a instrução *escrever (write),* +1109, obtém o número e o imprime (como um número inteiro de quatro dígitos e com sinal). A instrução *terminar (halt),* +4300, termina a execução.

Exemplo 2 Local	Número	Instrução
00	+1009	(Ler A)
01	+1010	(Ler B)
02	+200*	(Carregar A)

03	+3110	(Subtrair B)
04	+4107	(Desvio negativo para 07)
05	+1109	(Escrever A)
06	+4300	(Terminar)
07	+1110	(Escrever B)
08	+4300	(Terminar)
09	+0000	(Variavel A)
10	+0000	(Variavel B)

Esse programa em LMS lê dois números do teclado, determina o maior valor e o imprime. Observe o uso da instrução **+4107** como uma transferência condicional de controle, muito parecida com a instrução **if da** linguagem C. Agora escreva programas LMS para realizar as seguintes tarefas.
a) Use um loop controlado por um valor sentinela para ler 10 números positivos e calcular e imprimir sua soma.
b) Use um loop controlado por contador para ler sete números, alguns positivos e outros negativos, e calcule e imprima sua média.
c) Leia uma série de números e determine e imprima o maior deles. O primeiro número lido indica quantos números devem ser processados.

7.19 *(Um Simulador de Computador)* **À** primeira vista pode parecer chocante, mas neste problema você vai construir seu próprio computador. Não, você não estará unindo componentes. Em vez disso, você usará a poderosa técnica de *simulação baseada em software* para criar um *modelo de software* do Simpletron. Você não ficará desapontado. Seu simulador do Simpletron transformará seu computador em um Simpletron e você poderá realmente executar, testar e depurar os programas em LMS escritos no Exercício 7.18. Quando seu simulador Simpletron for executado, ele deve começar imprimindo:
***** Bem vindo ao Simpletron! *****
***** Por favor digite uma instrução (ou palavra *****
***** de dados) de seu programa por vez. Digitarei o *****
***** numero da posição e um ponto de interrogação *****
***** (?). Digite então a palavra para aquela posição *****
***** Digite o valor sentinela -99999 para encerrar a *****
***** digitação de seu programa. *****
Simule a memória do Simpletron com um array unidimensional **memória** com 100 elementos. Agora admita que o simulador está sendo executado, e vamos examinar o diálogo quando entrarmos com o programa do Exemplo 2 do Exercício 7.18:
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
***** Carregamento do programa concluido *** *** Inicio da execução do programa *****

O programa em LMS foi colocado (ou carregado) no array **memória.** Agora o Simpletron executa seu programa em LMS. A execução começa com a instrução no local **00** e, como o C, continua seqüencialmente, a menos que seja dirigido para alguma outra parte do programa por meio de uma transferência de controle

Use a variável **acumulador** para representar o registro acumulador. Use a variável **contadorInstrucao** para controlar o local da memória que contém a instrução que está sendo realizada. Use a variavel **codigoOperacao** para indicar a operação que está sendo realizada atualmente, i.e., os dois dígitos da esquerda, na palavra de instrução. Use a variável **operando** para indicar o local na memória no qual a instrução atual é aplicada. Dessa forma, **operando** compreende os dois dígitos da direita da instrução que está sendo realizada atualmente. Não execute instruções diretamente da memória. Em vez disso, transfira a próxima instrução a ser realizada da memória para a variável chamada **registrolnstrucao.** A seguir, "apanhe" os dois dígitos da esquerda e os coloque em **codigoOperacao** e "apanhe" os dois dígitos da direita e coloqueos em **operando.**

Quando o Simpletron iniciar a execução, os registros especiais são inicializados como se segue:

acumulador +0000
contadorlnstrucao 00
registrolnstrucao +0000
codigoOperacao 00
operando 00

Agora vamos "percorrer" a execução da primeira instrução LMS, +100 9 no local 00 da memória. Isso é chamado *ciclo de execução de instruções.*

O contadorlnstrucao nos informa o local da próxima instrução a ser realizada. Vamos *buscar* o conteúdo de tal local da memória usando a instrução C

registrolnstrucao = memória[contadorlnstrucao];

O código da operação e o operando são extraídos do registro de instruções pelas instruções

codigoOperacao = registrolnstrucao / 100; operando = registrolnstrucao *%* 100;

Agora o Simpletron deve determinar que o código da operação é realmente *ler* (e não *escrever, carregar* etc). Uma estrutura switch reconhece a diferença entre as doze operações da LMS.

Na estrutura switch, o comportamento das várias instruções LMS é simulado da maneira que se segue (deixamos as outras a cargo do leitor):

ler. scanf("%d", &memoria[operando]);
carregar. acumulador = memória[operando];
adicionar. acumulador += memória[operando];

Várias instruções de desvios: Serão vistas brevemente. *terminar:* Esta instrução imprime a mensagem

*** Execução do Simpletron concluida ***

e então imprime o nome e o conteúdo de cada registro assim como o conteúdo completo da memória. Tal saída é chamada freqüentemente *despejo (descarga* ou *dump) do computador* (não, um despejo de computador não é jogar fora um computador velho). Para ajudá-lo a programar a função de despejo, o formato de um exemplo de despejo é mostrado na Fig. 7.32. Observe que um despejo depois da execução de um programa Simpletron mostraria os valores reais das instruções e os valores dos dados no momento em que a execução terminasse.

Vamos continuar com a execução da primeira instrução de nosso programa, ou seja, +1009 no local 00. Como indicamos, a instrução switch simula isso realizando a instrução em C

scanf(***"%d",*** &memoria[operando]);

Deve ser mostrado um ponto de interrogação (?) na tela antes de scanf ser executado, para pedir a digitação do usuário. O Simpletron espera que o usuário digite um valor e então pressione a tecla *Return.* O valor é então lido e colocado no local 09.

```
REGISTROS:
acumulador +0000
contadorlnstrucao 00
registrolnstrucao +0000
codigoOperacao 00
operando 00

MEMÓRIA:
         0      1      2      3      4      5      6      7      8      9
 0   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
10   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
20   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
30   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
40   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
50   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
60   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
70   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
80   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
90   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
```

Fig. 7.32 Um exemplo de despejo.

Nesse momento termina a simulação da primeira instrução. Tudo que resta é preparar o Simpletron para executar a próxima instrução. Como a instrução que acabou de ser realizada não foi uma transferencia de controle, precisamos apenas incrementar o registro do contador de instruções como se segue:

++contadorInstrução;

Isso completa a execução simulada da primeira instrução. O processo inteiro (i.e., o ciclo de execução instruções) começa de novo com a busca da próxima instrução a ser executada.

Agora vamos examinar como as instruções de desvios — ou transferências de controle — são simuladas Tudo que precisamos fazer é ajustar apropriadamente o valor no contador de instruções. Assim, a instrução de desvio incondicional (4 0) é simulada dentro da estrutura switch da seguinte forma

contadorlnstrucao = operando;

A instrução condicional "desvio se o acumulador for zero" é simulada por

if(acumulador == 0)
contadorlnstrucao = operando;

Neste ponto você deve implementar seu simulador Simpletron e executar cada um dos programas LMS escritos no Exercício 7.18. Você deve aprimorar a LMS com recursos adicionais e fornecê-los ao seu simulador.

Seu simulador deve verificar vários tipos de erros. Durante a fase de carregamento do programa, por exemplo, cada número que o usuário digitar na memória do Simpletron deve estar no intervalo -9999 a + 9999. Seu simulador deve usar um loop while para examinar se cada número fornecido está nesse intervalo e, se não estiver, continuar a pedir ao usuário que forneça novamente o número até que seja fornecido um número correto.

Durante a fase de execução, seu simulador deve verificar vários erros sérios, como uma tentativa de divisão por zero, tentativa de executar códigos de operações inválidas, overflow do acumulador (i.e., operações aritméticas resultando em valores maiores do que +9999 ou menores do que -9999) e coisas assim. Tais

erros sérios são chamados *erros fatais.* Ao ser detectado um erro fatal, seu simulador deve imprimir uma mensagem de erro como:

*** Tentativa de dividir por zero ***
*** Interrupção anormal da execução do Simpletron ***
e deve um imprimir um despejo completo de computador no formato analisado anteriormente. Isso ajudará o usuário a localizar o erro no programa.

7.20 Modifique o programa de embaralhamento e distribuição de cartas da Fig. 7.24 de modo que as operações embaralhamento e distribuição sejam realizadas pela mesma função (embaralharEDistribuir). A função deve conter uma estrutura de loops aninhados similar à da função embaralhar da Fig. 7.24.

7.21 O que faz o seguinte programa?

```
#include <stdio.h>
void mystery1(char *,  const char *);
main() {
char stringl[80],  string2[80]; printf("Digite duas strings: "); scanf("%s%s",  stringl,  string2); mystery1(stringl, string2);
printf ("'%s\n", string1); return 0;
}
void mysteryl(char *s1,  const char *s2) {
while  (*s1  != '\0') + + s1;
for  (  ;   *s1 = *s2;  s1++, s2++) ;  /* instrução vazia */
```

7.22 O que faz o seguinte programa? **#include <stdio.h>**

```
int mystery2(const char *);
main()
char string[80]; ,
printf("Digite uma string: "); scanf("%s", string); printf ("9&d\n", mystery2 (string) ) ;
return 0;
int mystery2(const char *s)
int x = 0;
for  (   ;   *s   !=   '\0'; s++) + +x;
return x;
```

7.23 Encontre o erro em cada um dos seguintes segmentos de programas. Se o erro puder ser corrigido, explique como.

a) **int *number;**
printf("%d\n", *number);
b) **float *realPtr; long *integerPtr; integerPtr = realPtr;**
c) **int * x, y; x = y;**
d) **char s[]   = "isso e um array de caracteres"; int count;**
for (; *s !=   ' \0'; s + +) printf("%c ", *s);
e) **short *numPtr, result; void *genericPtr = numPtr; result = *genericPtr + 7;**
f) **float x = 19.34; float xPtr = &x; printf (**"5sf**n", xPtr) ;**
g) **char *s;**
printf("%s\n", s) ;

7.24 *(Classificação Rápida)* Nos exemplos e exercícios do Capítulo 6, analisamos as técnicas de classificação de bolhas, classificação de depósitos e classificação de seleção. Apresentamos agora uma técnica de classificação recursiva chamada *Classificação Rápida (Quicksort).* O algoritmo básico para um array unidimensional de valores é o seguinte:

1) *Etapa de Partição:* Tome o primeiro elemento do array não-ordenado e determine seu local

final no array classificado. Isso ocorre quando todos os valores à esquerda do elemento no array forem menores do que ele e todos os valores à direita do elemento no array forem maiores do que ele. Agora temos um elemento em seu local adequado e dois subarrays não-ordenados.

2) *Etapa de Recursão:* Realizar a etapa 1 em cada subarray não-ordenado.

Cada vez que a etapa 1 é realizada em um subarray, outro elemento é colocado em sua posição final no array ordenado, e dois subarrays não-ordenados são criados. Quando um subarray consistir em um elemento, ele deve ser ordenado, portanto aquele elemento está na sua posição final.

O algoritmo básico parece muito simples, mas como determinar a posição final do primeiro elemento de cada subarray? Como exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de partição — ele será colocado em sua posição final no array ordenado):

37 2 6 4 89 8 10 12 68 45

1) Iniciando com o elemento da extremidade esquerda do array, compare cada elemento com **37** até que seja encontrado um elemento menor do que **37,** então permute **37** e esse elemento. O primeiro elemento menor do que **37** é 12, portanto **37** e 12 são permutados. O novo array é:

12 2 6 4 89 8 10 **37** 68 45

O elemento 12 está em itálico para indicar que acabou de trocar de lugar com **37.**

2) Começando pela esquerda do array, mas com o elemento depois do 12, compare cada elemento **com 3** até encontrar um elemento maior do que **37,** e então permute **37** com esse elemento. O primeiro elemento maior do que **37** é 89, portanto **37** e 89 são permutados. O novo array é:

12 2 6 4 **37** 8 10 *89* 68 45

3) Começando pela direita, mas com o elemento antes de 89, compare cada elemento com **37** até que seja encontrado um elemento menor do que **37,** e então permute **37** com esse elemento. O primeiro elemento menor do que **37** é 10, portanto **37** e 10 são permutados. O novo array é

12 2 6 4 *10* 8 **37** 89 68 45

4) Começando pela esquerda, mas com o elemento após o 10, compare cada elemento com **37** até **que seja** encontrado um elemento maior do que **37,** e então permute **37** com esse elemento. Não há mais tos maiores do que **37,** portanto ao compararmos **37** com ele mesmo sabemos que **37** foi colocado em sua posição final no array ordenado.

Depois de a partição ter sido aplicada no array anterior, ficam-se com dois arrays não-ordenados. O subarray com valores menores do que 37 contém 12, 2, 6,4, 10 e 8. O subarray com valores maiores do que 37 **contêm** 89, 68 e 45. A ordenação continua da mesma maneira que no array original.

Com base na análise anterior, escreva uma função recursiva quicksort para ordenar um array inteiro unidimensional. A função deve receber como argumentos um array inteiro, um subscrito inicial e um subescrito final. A função partição deve ser chamada por quicksort para realizar a etapa de partição.

7.25 **7.25** *(Travessia de Labirinto)* A grade de uns e zeros a seguir é uma representação bidimensional de um labirinto.

111111111111 100010000001 001010111101 111010000101 100001110100 111101010101 100101010101 110101010101 100000000101 111111011101 100000010001 111111111111

Os uns representam as paredes do labirinto, e os zeros representam os quadrados (espaços) dos ca; possíveis para cruzar o labirinto.

Há um algoritmo simples para atravessar um labirinto que garante a localização da saída (admitindo que há uma saída). Se não houver uma saída, você chegará novamente ao local de partida. Coloque sua mão direita na parede de sua direita e comece a andar para a frente. Nunca tire sua mão direita da parede Se o labirinto dobrar para a direita, siga a parede da direita. Contanto que sua mão direita não seja retirada da parede, você acabará chegando à saída. Pode haver um caminho mais curto, mas esse garante **sua saída do labirinto** Escreva uma função

recursiva travessiaLabirinto para atravessar um labirinto. A função deve receber como argumentos um array 12-por-12 de caracteres representando o labirinto e um local de **p:** A medida que a função travessiaLabirinto tentar localizar uma saída do labirinto, ela deve colocar um caractere X em cada quadrado do caminho percorrido. A função deve exibir o labirinto depois de cada movimento para que o usuário possa ver como sua saída será encontrada.

7.26 *(Gerando Labirintos Aleatoriamente)* Escreva uma função geradorLabirinto que tome um array 12-por-12 de caracteres como argumento e produza aleatoriamente um labirinto. A função também deve fornecer os locais de partida e de saída do labirinto. Experimente sua função travessiaLabirinto do Exercício 7.25 usando vários labirintos gerados aleatoriamente.

7.27 *(Labirintos de Qualquer Tamanho)* Generalize as funções **travessiaLabirinto** e **geradorLabirinto** dos Exercícios 7.25 e 7.26 para processar labirintos de qualquer largura e altura.

7.28 *(Arrays de Ponteiros a Funções)* Reescreva o programa da Fig. 6.22 para usar uma interface baseada em menus. O programa deve oferecer ao usuário as 4 opções a seguir:

```
Digite uma escolha
0 Imprimir o array de graus
1 Encontrar o grau minimo
2 Encontrar o grau maximo
3 Imprimir a media de todos os testes de cada aluno
4 Finalizar o programa
```

Uma restrição ao uso de arrays de ponteiros a funções é que todos os ponteiros devem ter o mesmo tipo. Os ponteiros devem ser para funções que tenham o mesmo tipo de retorno e que recebam argumentos do mesmo tipo. Por esse motivo, as funções da Fig. 6.22 devem ser, modificadas de forma que cada uma delas retorne o mesmo tipo e utilize os mesmos parâmetros. Modifique as funções **minimo** e **máximo** para imprimir os valores mínimo ou máximo e nada retornar. Para a opção 3, modifique a função **media** da Fig. 6.22 para imprimir a média de cada aluno (e não de um aluno determinado). A função **media** não deve retornar nada e utilizar os mesmos parâmetros de **imprimirArray, minimo** e **máximo.** Armazene os ponteiros para as quatro funções no array **processarGraus** e use a opção feita pelo usuário como subscrito do array na chamada de cada função.

7.29 *(Modificações no Simulador Simpletron)* No Exercício 7.19, você escreveu uma simulação de software que executa programas escritos na Linguagem de Máquina Simpletron (LMS). Neste exercício, propomos várias modificações e melhoramentos no Simulador Simpletron. Nos Exercícios 12.26 e 12.27, propomos a construção de um compilador que converta programas escritos em linguagem de programação de alto nível (uma variação do BASIC) para a Linguagem de Máquina Simpletron. Algumas das modificações e melhoramentos a seguir podem ser necessários para executar os programas produzidos pelo compilador, a) Amplie a memória do Simulador Simpletron para conter 1000 posições de memória de forma a permitir ao Simpletron manipular programas maiores. b) Permita ao simulador realizar o cálculo de resto de divisões (modulus). Isso exige uma instrução adicional da Linguagem de Máquina Simpletron.
c) Permita ao simulador realizar cálculos de potências (exponenciação). Isso exige uma instrução adicional da Linguagem de Máquina Simpletron.
d) Modifique o simulador para usar valores hexadecimais em vez de valores inteiros para

representar instruções da Linguagem de Máquina Simpletron.
e) Modifique o simulador para permitir a saída de uma nova linha. Isso exige uma instrução adicional da Linguagem de Máquina Simpletron.
f) Modifique o simulador para processar valores em ponto flutuante além dos valores inteiros.
g) Modifique o simulador para manipular a entrada de strings. Sugestão: Cada palavra do Simpletron pode ser dividida em dois grupos, cada um deles contendo um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente decimal ASCII de um caractere. Adicione uma instrução em linguagem de máquina que entre com uma string e armazene-a iniciando em um local específico da memória do Simpletron. A primeira metade da palavra nesse local será o valor do número de caracteres da string (i.e., o comprimento da string). Cada meia palavra a seguir contém um caractere ASCII expresso como dois dígitos decimais ASCII. A instrução em linguagem de máquina converte cada caractere em seu equivalente ASCII e o atribui à meia palavra.
h) Modifique o simulador para manipular a saída de strings armazenadas no formato da parte (g). Sugestão: Adicione uma instrução em linguagem de máquina que imprima uma string iniciando em uma determinada posição da memória do Simpletron. A primeira metade da palavra nessa posição é o valor do número de caracteres da string (i.e., o comprimento da string). Cada meia palavra a seguir contém um caractere ASCII expresso por dois dígitos decimais. A instrução em linguagem de máquina examina o comprimento e imprime a string traduzindo cada número de dois dígitos em seu caractere equivalente.

7.30 O que faz o seguinte programa?

```
#include <stdio.h>
int mystery3(const char *, const char *); main( )
{
char stringl[80],  string2[80];
printf("Digite duas strings: "); scanf("%s%s",  stringl, string2);
printf("O resultado e %d\n", mystery3(string1, string2)); return 0;
}
int mystery3(const char *s1,  const char *s2) {
for (   ;   *s1   !=   '\0'  && *s2   !=   '\0';   s1++, s2++)
if   (*s1  != *s2) return 0;
return 1;
}
```

8

Caracteres e Strings

Objetivos

- Ser capaz de usar as funções da biblioteca de manipulação de caracteres **(ctype).**
- Ser capaz de usar as funções de entrada/saída de strings e caracteres da biblioteca-padrão de entrada/saída **(Stdio**).
- Ser capaz de usar as funções de conversão de strings da biblioteca geral de utilitários**(stdlib)**.
- Ser capaz de usar as funções de processamento de strings da biblioteca de manipuação de strings **(s**trin**g).**
- Avaliar o poder das bibliotecas de funções como um meio de conseguir a reutilização te software.

O principal defeito de Henry King
Era *mascar pequenas quantidades de palavras.*
Hilaire Belloc

Adapte os atos às palavras, as palavras aos atos.
William Shakespeare

A escrita enérgica é concisa.
Uma frase não deve conter palavras desnecessárias,
um parágrafo não deve conter frases desnecessárias.
Willian Strunk. Jr.

Em uma concatenação apropriada.
Oliver Goldsmith

Sumário

8.1 Introdução

Neste capítulo, apresentamos as funções da biblioteca-padrão do C que facilitam o processamento de strings e caracteres. As funções permitem aos programas processarem caracteres, strings, linhas de texto e blocos de memória.

Este capítulo analisa as técnicas usadas para desenvolver editores, processadores de textos, layout de páginas, sistemas computadorizados de composição e outros tipos de software de processamento de texto. As manipulações de texto realizadas pelas funções de entrada/saída formatadas como **printf** e **scanf** podem ser implementadas por meio das funções analisadas neste capítulo.

8.2 Conceitos Fundamentais de Strings e Caracteres

Os caracteres são as peças fundamentais dos programas-fonte. Cada programa é composto de uma sequência de caracteres que — ao serem agrupados de uma forma significativa — são interpretados pelo computador como uma série de instruções usadas para realizar uma tarefa. Um programa pode **conter** ***constantes de caracteres.*** Uma constante de caractere é um valor **int** representado como um caractere entre aspas simples. O valor da constante de caracteres é o valor inteiro do caractere no conjunto de caracteres do equipamento. Por exemplo, 'z' representa o valor inteiro do **z,** e '**\n**' representa o v**alor** inteiro de uma nova linha.

Uma string é uma série de caracteres tratada como uma unidade simples. Uma string pode incluir letras, dígitos e vários ***caracteres especiais*** como +, -, *, /, $ e outros. ***Strings literais*** ou consta***ntes strings (constantes alfanuméricas)*** são escritas em C entre aspas duplas, como se segue:

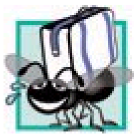
"Paulo P. Silva" (um nome)
"Rua Grande 9999" (um endereço)
"Salvador, Bahia" (uma cidade e um estado)
"(071) 555-1212" (um número de telefone)

Uma string em C é um array de caracteres que termina com o ***caractere NULL (***'\0'). Uma string é acessada por meio de um ponteiro para seu primeiro caractere. O valor de uma string é o endereço de seu primeiro caractere. Dessa forma, em C é apropriado dizer que ***uma string é um ponteiro*** — na realidade um ponteiro para o primeiro caractere da string. Nesse sentido, as strings são como arrays, por que um array também é um ponteiro para seu primeiro elemento.

Uma string pode ser atribuída em uma declaração para um array de caracteres ou para uma variável do tipo char ***.** Cada uma das declarações

```
char cor[]  = "azul";
char *corPtr = "azul";
```

inicializa uma variável com a string "**azul".** A primeira declaração cria um array de 5 elementos **cor contendo** os caracteres '**a', 'z', 'u', 'l'** e '**0'. A** segunda declaração cria uma variável ponteiro corPtr que aponta para a string "**azul"** em algum lugar da memória.



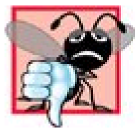
Dicas de portabilidade 8.1

Quando uma variável do tipo ***char*** ** for inicializado com uma string literal, alguns compiladores podem coloc**ar** a string em um local da memória onde ela não pode ser modificada. Se for necessário modificar uma string, ela deve ser armazenada em um array de caracteres para que seja assegurada a possibilidade de modificá-la em todos os sistemas.*

A declaração do array anterior também poderia ter sido escrita

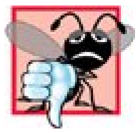
char cor[] = {'a','z','u','l','\0'};

Ao declarar um array de caracteres para conter uma string, o array deve ser suficientemente grande para armazenar a string e seu caractere **NULL** de terminação. A declaração anterior determina automaticamente o tamanho do array baseado no número de valores na lista de inicializadores.



Erro comun de programação 8.1

*Não alocar espaço suficiente em um array de caracteres para armazenar o caractere **NULL** que termina uma string.*



Erro comun de programação 8.2

*Imprimir uma "string" que não contém um caractere **NULL** de terminação.*



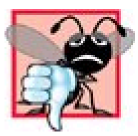
Boa prática de programação 8.1

Ao armazenar uma string de caracteres em um array, certifique-se de que o array é suficientemente grande para conter a maior string que será armazenada. A linguagem C permite que sejam armazenadas strings de qualquer comprimento. Se uma string for maior do que o array de caracteres no qual está sendo armazenada, os caracteres além do final do array irão sobrescrever os dados da memória após o array.

Uma string pode ser atribuída a um array usando **scanf.** Por exemplo, a instrução a seguir atribui uma string ao array de caracteres w**ord[20];**

scanf("%s", word);

A string fornecida pelo usuário é armazenada em **word** (observe que **word** é um array que, obviamente, é um ponteiro, de forma que & não é necessário com o argumento **word).** A função **scanf lerá** os caracteres até que um espaço ou um indicador de nova linha ou de fim de linha seja encontrado. Observe que a string não deve ter comprimento maior do que 19 caracteres para deixar espaço para o caractere NULL de terminação. Para que um array de caracteres seja impresso como uma string, o array deve conter um caractere **NULL** de terminação.



Erro comun de programação 8.3

Processar um caractere simples como uma string. Uma string é um ponteiro — provavelmente um inteiro grande considerável. Entretanto, um caractere é um inteiro pequeno (valores ASCII variando de 0 a 255). Em muitos sistemas, isso causa um erro porque os endereços pequenos da memória são reservados para finalidades especiais como handlers de interrupção do sistema operacional — dessa forma, ocorrem "violações de acesso".

Erro comun de programação 8.4

Passar um caractere como argumento para uma função quando uma string é esperada.

Erro comun de programação 8.5

Passar uma string como argumento para uma função quando um caractere é esperado.

8.3 Biblioteca de Manipulação de Caracteres

A biblioteca de manipulação de caracteres inclui várias funções que realizam testes úteis e manipulações de dados de caracteres. Cada função recebe um caractere — representado como um int -- ou EOF como argumento. Como vimos no Capítulo 4, freqüentemente os caracteres são manipulados como inteiros porque um caractere em C é um inteiro de um byte. Lembre-se de que normalmente EOF tem o valor — 1 e algumas arquiteturas de hardware não permitem que valores negativos sejam armazenados em variáveis char. Portanto, as funções de manipulação de caracteres tratam os caracteres como inteiros. A Fig. 8.1 oferece um resumo das funções da biblioteca de manipulação de caracteres.



Boa prática de programação 8.2

*Ao usar funções da biblioteca de manipulação de caracteres, inclua o arquivo de cabeçalho **<ctype.h>.***

O programa da Fig. 8.2 demonstra as funções ***isdigit,, isa1pha, isa1num e isxdigit. A*** função isdigit determina se seu argumento é um dígito (0-9). A função isalpha determina se seu argumento é uma letra maiúscula **(A-Z)** ou minúscula **(a — z).** A função **isa1num** determina se seu argumento é uma letra maiúscula, uma letra minúscula ou um dígito. A função is**xdigit determina** se seu argumento é um dígito hexadecimal **(A — F, a — f, 0 — 9).**

Protótipo	Descrição da função
int isdigit(int c)	Retoma um valor verdadeiro se **c** for um dígito e 0 (falso) em caso contra:
int isalpha(int c)	Retorna um valor verdadeiro se **c** for uma letra e 0 em caso contrário.
int isalnum(int c)	Retorna um valor verdadeiro se **c** for um dígito ou uma letra e 0 em caso contrário.
int isxdigit(int c)	Retorna um valor verdadeiro se **c** for um caractere de dígito hexadecimal e em caso contrário. (Veja o Apêndice D, "Sistemas de Numeração", para c: uma explicação detalhada dos números binários, decimais e hexadecimais.)
int islower(int c)	Retorna um valor verdadeiro se **c** for uma letra minúscula e 0 em caso contrário.
int isupper(int c)	Retorna um valor verdadeiro se **c** for uma letra maiúscula e 0 em caso contrário.
int tolower(int c)	Se **c** for uma letra maiúscula, **tolower** retorna **c** como uma letra minúscula. Caso contrário, **tolower** retorna o argumento inalterado.
int toupper(int c)	Se **c** for uma letra minúscula, **toupper** retorna **c** como uma letra maiúscula. Caso contrário, **toupper** retorna o argumento inalterado.
int isspace(int c)	Retorna um valor verdadeiro se **c** for um caractere de espaço em branco — nova linha (' **\n**'), espaço (‘ ‘), avahço de folha (' **\f** '), carriage return **(' \r** '), tabulação horizontal (' **\t**') ou tabulação vertical (' **\v**') — e 0 em caso contrário.

int iscntrl(int c)	Retoma um valor verdadeiro se **c** for um caractere de controle e 0 em caso 1 contrário.
int ispunct(int c)	Retoma um valor verdadeiro se **c** for um caractere imprimivel diferente de . espaço, um dígito ou uma letra e 0 em caso contrário.
int isprint(int c)	Retoma um valor verdadeiro se **c** for um caractere imprimivel incluindo espaço (' ') e 0 em caso contrário.
int isgraph(int c)	Retorna um valor verdadeiro se **c** for um caractere imprimivel diferente de espaço (' ') e 0 em caso contrário.

Fig 8.1 Resumo das funções da biblioteca de manipulação de caracteres

```
/*Usando as funções isdigit,  isalpha,  isalnum e isxdigit */
#include <stdio.h>
#include <ctype.h>

main(){
printf("%s\n%s%s\n%s%s\n\n","De acordo com isdigit: ",
         isdigit('8') ? "8 e um " : "8 nao e um ", "digito",
         isdigit('#' )  ? "# e um "   :  "# nao e um ", "digito");

printf ("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",   "De acordo com isalpha: ",
          isalpha('A')   ?  "A e uma "   :   "A nao e uma ", "letra",
          isalpha('b')   ?  "b e uma "   :   "b nao e uma ", "letra",
          isalpha('&')  ? "& e uma "   :  "& nao e uma ", "letra",
          isalpha('4')   ?  "4 e uma "   :   "4 nao e uma ", "letra");

printf("%n%s%s\n%s%s\n%s%s\n\n",   "De acordo com isalnum: ",
        isalnum('A') ? "A e um " : "A nao e um ", "digito ou uma letra",
        isalnum('8') ? "8 e um " : "8 nao e um ", "digito ou uma letra",
        isalnum('#')   ?  "# e um "   :   "# nao e um ",   "digito ou uma letra");

printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n", "De acordo com isxdigit:",
        isxdigit('F') ? "F e um " : "F nao e um ", "digito hexadecimal",
        isxdigit('J') ? "J e um " : "J nao e um ", "digito hexadecimal",
        isxdigit('7') ? "7 e um " : "7 nao e um ", "digito hexadecimal",
        isxdigit('$') ? "$ e um " : "$ nao e um ", "digito hexadecimal",
        isxdigit('f') ? "f e um " : "f nao e um ", "digito hexadecimal");

return 0;
}
```

```
De acordo com isdigit:
8 e um digito
# nao e um digito

De acordo com isalpha:
A e uma letra
b e uma letra
& nao e uma letra
4 nao e uma letra

De acordo com isalnum:
A e um digito ou uma letra
8 e um digito ou uma letra
# nao e um digito ou uma letra

De acordo com isxdigit:
F e um digito hexadecimal
J nao e um digito hexadecimal
7 e um digito hexadecimal
$ nao e um digito hexadecimal
f e um digito hexadecimal
```

Fig. 8.2 Usando **isdigit**, **isalpha**, **isalnum**, e **isxdigit.**

```
/* Usando as funções islower,  isupper, tolower e toupper */
#include <stdio.h>
#include <ctype.h>
main() {

printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "De acordo com islower:",
        islower('p')  ? "p e uma "   :  "p nao e uma ", "letra minúscula",
        islower('P')   ?  "P e uma "   :   "P nao e uma ", "letra minúscula",
        islower('5')   ?  "5 e uma "   :   "5 nao e uma ", "letra minúscula",
        islower('!')   ?  "!  e uma "   :   "! nao e uma ", "letra minúscula");

printf ("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "De acordo com isupper:",
        isupper('D')  ?  "De uma "   :   "D nao e uma ", "letra maiúscula",
        isupper('d')  ?  "de uma "   :   "d nao e uma ", "letra maiúscula",
        isupper('8')  ?  "8 e uma "  :   "8 nao e uma ", "letra maiúscula",
        isupper('$')  ?  "$ e uma "  :   "$ nao e uma ", "letra maiúscula");

printf("%s%c\n%s%c\n%s%c\n%s%c\n",
        "u convertido para letra maiúscula e ", toupper('u'),
        "7 convertido para letra maiúscula e ", toupper('7'),
        "$ convertido para letra maiúscula e ", toupper('$'),
        "L convertido para letra minúscula e ",  tolower('L'));

return 0;
}
```

```
De acordo com islower:
p e uma letra minúscula
P nao e uma letra minúscula
5 nao e uma letra minúscula
! nao e uma letra minúscula

De acordo com isupper:
D e uma letra maiúscula
d nao e uma letra maiúscula
8 nao e uma letra maiúscula
$ nao e uma letra maiúscula
u convertido para letra maiúscula e U
7 convertido para letra maiúscula e 7
$ convertido para letra maiúscula e $
L convertido para letra minúscula e 1
```

Fig. 8.3 Usando **islower, isupper, tolower e toupper.**

O programa da Fig. 8.2 usa o operador condicional (?:) com cada função para determinar se a string " e **um "** ou a string " na**o e um "** deve ser impressa na saída de cada caractere examinado. Por exemplo, a expressão

isdigit('8') ? " 8 e um " : "8 nao e um "

indica se '**8'** for um dígito, i.e., se **isdigit** retornar um valor verdadeiro (diferente de zero), a string " **8 e um "** é impressa e se '8' não for um dígito, i.e., se **isdigit** retornar **0,** a string " **8 nao e um "** é impressa.

O programa da Fig. 8.3 demonstra as funções ***islower, isupper, tolower e toupper.*** **A** função **islower** determina se seu argumento é uma letra minúscula **(a — z). A** função i**supper** determina se seu argumento é uma letra maiúscula **(A-Z). A** função **tolower** converte uma letra maiúscula em uma letra minúscula e retorna a letra minúscula. Se o argumento não for uma letra maiúscula, to**lower** retorna o argumento inalterado. **A** função **toupper** converte uma letra minúscula em uma letra maiúscula e retorna a letra maiúscula. Se o argumento não for uma letra minúscula, **toupper** retorna o argumento inalterado.

A Fig. 8.4 demonstra as funções ***isspace, iscntrl, ispunct, isprint*** e ***isgraph.*** **A** função iss**pace** determina se seu argumento é um dos seguintes caracteres de espaço em branco: espaço (' '), avanço de folha ('**\f'),** nova linha ('**\n'),** carriage return ('**\r'),** tabulação horizontal ('**\t'),** ou tabulação vertical ('\v'). A função is**cntrl** determina se seu argumento é um dos seguintes caracteres de controle: tabulação horizontal ('**\t'),** tabulação vertical ('\v'), avanço de folha ('\f'), alerta ('**\a'),** backspace ('\b'), carriage return ('\r') ou nova linha ('\n'). A função **ispunct** determina se seu argumento é um caractere imprimível diferente de um espaço, um dígito ou uma letra, como $,#, (,), [,], {, }, ;, :, % etc. A função **isprint** determina se seu argumento é um caractere, que pode ser exibido na tela (incluindo o caractere de espaço). A função is**graph** verifica os mesmos caracteres que **isprint,** entretanto, o caractere de espaço não está incluído.

```
/* Usando as funções isspace,  iscntrl,  ispunct,  isprint,  isgraph */
#include <stdio.h>
#include <ctype.h>

main(){

printf("%s\n%s%s%s\n%s%s%s\n%s%s\n\n",   "De acordo com isspace:",
            "Nova linha",  isspace('\n')  ?  " e um "   :   " nao e um ",
            "caractere de espaço em branco",   "Tabulação horizontal",
            isspace('\t')  ?  " e um "   :   " nao e um ",
            "caractere de espaço em branco",
            isspace('%')  ?  "% e um "   :   "% nao e um ",
            "caractere de espaço em branco");

printf("%s\n%s%s%s\n%s%s\n\n",   "De acordo com iscntrl:",
            "Nova linha",  iscntrl('\n')  ?  " e um "   :   " nao e um ",
            "caractere de controle ",  iscntrl('$')  ?  "$ e um "   :   "$ nao e um ",
            "caractere de controle");

printf("%s\n%s%s\n%s%s\n%s%s\n\n",   "De acordo com ispunct:",
            ispunct(';')  ?  ";  e um "   :   "; nao e um ",
            "caractere de pontuação",
            ispunct('Y')  ?  "Y e um "   :   "Y nao e um ", "caractere de pontuação",
            ispunct('#')  ?  "# e um "   :   "# nao e um ",
            "caractere de pontuação");

printf ("%s\ns&s%s\n%s%s%s\n\n",   "De acordo   com isprint:",
            isprint('$')  ?  "$ e um "   :   "$ nao e um ",   "caractere imprimivel",
            "Alerta",  isprint('\a')  ? " e um "   :   " nao e um ",
            "caractere imprimivel");

printf("%s\n%s%s\n%s%s%s\n",   "De acordo com isgraph:",
            isgraph('Q')  ?  "Q e um "   :   "Q nao e um ",
            "caractere imprimivel diferente de um espaço",
            "Espaço",  isgraph('   ')  ? " e um "  :  " nao e um ",
            "caractere imprimivel diferente de um espaço");
return 0;
}
```

```
De acordo com isspace:
Nova linha e um caractere de espaço em branco
Tabulação horizontal e um caractere de espaço em branco
% nao e um caractere de espaço em branco

De acordo com iscntrl:
Nova linha e um caractere de controle
$ nao e um caractere de controle

De acordo com ispunct:
; e um caractere de pontuação
Y nao e um caractere de pontuação
# e um caractere de pontuação

De acordo com isprint:
$ e um caractere imprimivel
Alerta nao e um caractere imprimivel

De acordo com isgraph:
Q e um caractere imprimivel diferente de um espaço
Espaço nao e um caractere imprimivel diferente de um espaço
```

Fig. 8.4 Usando **isspace, iscntrl, ispunct, isprint e isgraph.**

8.4 Funções de Conversão de Strings

Esta seção apresenta ***as, funções de conversão de strings*** da ***biblioteca geral de utilitários (stdlib).*** Essas funções convertem strings de dígitos em valores inteiros ou de ponto flutuante. A Fig. 8.5 oferecer um resumo das funções de conversão de strings. Observe o uso de **const** para declarar a variável nPtr nos cabeçalhos das funções (ler da direita para a esquerda como **"nPtr** é um ponteiro para um caractere constante"); **const** declara que o valor do argumento não será modificado.

Protótipo da função	Descrição da função
double atof(const char *nPtr)	Converte a string nPtr em double.
int atoi (const char *nPtr)	Converte a string nPtr em int.
long atol(const char *nPtr)	Converte a string nPtr em inteiro longo(long)
double strtod(const char *nPtr, char **endPtr)	Converte a string nPtr em double
Long strtol (const char *nPtr, char **endPtr, int base)	Converte a string nPtr em long.
unsigned long strtoul (const char *nPtr, char **endPtr, int base)	Converte a string nPtr em unsigned long

Fig. 8.5 Resumo das funções de conversão de strings da biblioteca geral de utilitários

```
/* Usando atof */
#include <stdio.h>
#include <stdlib.h>

main(){

double d;

d = atof("99.0");
printf("%s%.3f\n%s%.3f\n",
        "A string \"99.0\" convertida em double e ", d,
        "O valor convertido dividido por 2 e ", d / 2.0);

return 0;
}
```

```
A string "99.0" convertida em double e 99.000
O valor convertido dividido por 2 e 49.500
```

Fig. 8.6 Usando **atof.**



Boa prática de programação 8.3

*Ao usar funções da biblioteca geral de utilitários, inclua o arquivo de cabeçalho **<stdlib.h>.***

A função **atof** (Fig. 8.6) converte seu argumento — uma string que representa um número de ponto flutuante — em um valor **double.** A função retorna o valor **double.** Se o valor convertido não puder ser representado — por exemplo, se o primeiro caractere da string não for um dígito — o comportamento da função **atof** fica indefinido.

A função **atoi** (Fig. 8.7) converte seu argumento — uma string de dígitos que representam um inteiro — em um valor **int.** Se o valor convertido não puder ser representado, o comportamento da função **atoi** fica indefinido.

```
/* Usando atoi */
#include <stdio.h>
#include <stdlib.h>

main (){

int i;

i = atoi("2593");
printf("%s%d\n%s%d\n",
        "A string \"2593\" convertida em int e ", i,
        "0 valor convertido menos 593 e ",  i - 593);

return 0;
}
```

```
string "2593" convertida em int e 2593
EO valor convertido menos 593 e 2000
```

Fig. 8.7 Usando **atoi.**

```
/* Usando atoi */
#include <stdio.h>
#include <stdlib.h>

main() {

long 1;

1 = atoi("1000000");

printf ("%s%ld\n%s%ld\n",
          "A string \"1000000\" convertida em long int e ", 1,
          "O valor convertido dividido por 2 e ", 1/2);

return 0;
}
```

```
A string "1000000" convertida em long int e 1000000
O valor convertido dividido por 2 e 500000
```

Fig. 8.8 Usando **atol.**

A função **atol** (Fig. 8.8) converte seu argumento — uma string de dígitos que representam um inteiro longo — em um valor **long.** A função retorna o valor l**ong.** Se o valor convertido não puder ser representado, o comportamento da função **atol** fica indefinido. Se **int** e **long** estiverem armazenados em 4 bytes, as funções **atoi** e **atol** funcionam de maneira idêntica.

```
/* Usando strtod */
#include <stdio.h>
#include <stdlib.h>

main() {

double d;

char *string =  "51.2% foram admitidos";
char *stringPtr;

d = strtod(string,  &stringPtr);

printf("A string \"%s\"  foi convertida para o\n", string);
printf("valor double %.2f e a string \"%s\"\n", d,  stringPtr);

return 0;
}
```

```
A string "51.2% foram admitidos" foi convertida para o
valor double 51.20 e a string "% foram admitidos"
```

Fig. 8.9 Usando **strtod.**

```
/* Usando strtol */
#include <stdio.h>
#include <stdlib.h>


main(){
long x;
char *string =  "-1234567abc",  *remainderPtr;

x = strtol(string,  &remainderPtr, 0);

printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
        "A string original e ", string,
        "O valor convertido e ", x,
        "O restante da string original e ",
        remainderPtr,
        "O valor convertido mais 567 e ", x + 567);

return 0;
}
```

```
A string original e "-1234567abc"
O valor convertido e -1234567
O restante da string original e "abe"
S valor convertido mais 567 e -1234000
```

Fig. 8.10 Usando **strtol.**

A função ***strtod*** (Fig. 8.9) converte uma seqüência de caracteres que representam um valor de ponto flutuante em **double.** A função recebe dois argumentos — uma string **(char *)** e um ponteiro para uma string. A string contém a seqüência de caracteres a ser convertida em **double. O** ponteiro é atribuído ao local do primeiro caractere depois da parte convertida da string. A instrução

d = strtod(string, &stringPtr);

do programa da Fig. 8.9 indica que **d** recebe a atribuição do valor convertido de **string,** e **stringPtr recebe** a atribuição do local do primeiro caractere depois do valor convertido (51.2) em **string.** A função ***strtol*** (Fig. 8.10) converte em **long** uma seqüência de caracteres que representa um inteiro. A função recebe três argumentos — uma string **(char *),** um ponteiro para uma string e um inteiro. A string contém a seqüência de caracteres a ser convertida. O ponteiro recebe a atribuição do local do primeiro caractere depois da parte convertida da string. Um inteiro especifica a *base* do valor que está sendo convertido. A instrução

x = strtoul(string,&remainderPtr, 0);

no programa da Fig. 8.10 indica que x recebe a atribuição do valor **long** convertido de **string. O segund**o argumento, **remainderPtr,** recebe a atribuição do restante de **string** após a conversão. Usar **NULL** para o segundo argumento faz com

que o restante da string seja ignorado. O terceiro argumento, 0, indica que o valor a ser convertido pode estar no formato octal (base 8), decimal (base 10) ou hexadecimal (base 16). A base pode ser especificada como 0 ou qualquer valor entre 2 e 36. Veja o Apendice D, "Sistemas de Numeração" para obter uma explicação detalhada sobre os sistemas de numeração octal, decimal e hexadecimal. As representações numéricas de inteiros da base 11 à base 36 ***usam*** os caracteres A-Z para representar os valores 10 a 35. Por exemplo, os valores hexadecimais podem ser constituídos dos dígitos 0-9 e dos caracteres A-F. Um inteiro da base 11 pode ser constituído dos dígitos 0-9 e do caractere A. Um inteiro da base 24 pode ser constituído dos dígitos 0-9 e dos caracteres A-N. Um inteiro da base 36 pode ser constituído dos dígitos 0-9 e dos caracteres A-Z. A função ***strtoul*** (Fig. 8.11) converte **para unsigned long,** uma seqüência de caracteres que represente um inteiro u**nsigned long.** A função funciona de maneira idêntica à função **strtol.** A instrução

x = strtoul(string, &remainderPtr, 0);

no programa da Fig. 8.11 indica que x recebe a atribuição do valor **unsigned long** convertido da string. O segundo argumento. **&remainderPtr,** recebe a atribuição do restante de **string** depois da conversão. O terceiro argumento, 0, indica que o valor a ser convertido pode estar no formato octal, decimal ou hexadecimal.

```
/* Usando strtoul */
#include <stdio.h>
#include <stdlib.h>

main() {

unsigned long x;
char *string =  "1234567abc",  *remainderPtr;

x = strtoul(string,  &remainderPtr, 0);

printf("%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
          "A string original e ", string,
          "O valor convertido e ", x,
          "O restante da string original e ",
          remainderPtr,
          "O valor convertido menos 567 e ",  x - 567);

return 0;
}
```

```
A string original e "1234567abc"
O valor convertido e 1234567
O restante da string original e "abc"
S valor convertido mais 567 e 1234000
```

Fig. 8.11 Usando **strtoul.**

8.5 Funções da Biblioteca-padrão de Entrada/Saída

Esta seção apresenta várias funções da biblioteca-padrão de entrada/saída **(stdio)** especificamente para manipulação de dados de caracteres e strings. A Fig. 8.12 traz um resumo das funções de entrada/saída de caracteres e strings da biblioteca-padrão de entrada/saída.

Protótipo da função	Descrição da função
int getchar (void)	Obtém o próximo caractere do dispositivo-padrão de entrada e retorna como inteiro.
char *gets (char *s)	Obtém caracteres do dispositivo-padrão de entrada para o array **s** até que um caractere de nova linha ou de fim de arquivo seja encontrado. Um caractere de terminação **NULL** é adicionado ao array.
int putchar(int c)	Imprime o caractere armazenado em c.
int puts(const char *s)	Imprime a string s seguida de um caractere de nova linha.
int sprint(char *s, const char *format, …)	Equivalente ao printf exeto que a saída é armazenada no array s em vez de ser impressa na tela.
int sscanf(char *s, const char *format, …_	Equivalente a scanf, exeto que a entrada é lida no array s em vez de ser lida no teclado.

Fig. 8.12 As funções de caracteres e strings da biblioteca-padrão de entrada/saída.



Boa prática de programação 8.4

*Ao usar funções da biblioteca-padrão de entrada/saída, inclua o arquivo de cabeçalho **<stdio.h>.***

O programa da Fig. 8.13 usa as funções ***gets*** **e put*char*** para ler uma linha de texto do dispositivo-padrão de entrada (teclado) **e** enviar recursivamente para o dispositivo de saída os caracteres de uma linha na ordem inversa. A função **gets** lê os caracteres do dispositivo-padrão de entrada para o seu argumento - um array do tipo **char** — até que um caractere de nova linha ou o indicador de fim de arquivo seja encontrado. Um caractere NULL ('\0') será adicionado ao array quando **a** leitura terminar. A função put**char** imprime seu argumento caractere. O programa chama **a** função recursiva **reverse** para imprimir a linha de texto na ordem inversa. Se o primeiro caractere do array recebido por **reverse** for o caractere NULL '\0', **reverse** retoma. Caso contrário, **a** função **reverse** é chamada novamente com o endereço do subarray que inicia no elemento **s [**1], **e** o caractere **s[** 0] é enviado ao dispositivo de saída com putchar quando **a** chamada recursiva for concluída. A ordem das duas instruções na parte **else** da estrutura if faz com que vá para o caractere NULL de terminação da string antes de o caractere ser impresso. À medi**da** que **as** chamadas recursivas são concluídas, os caracteres são impressos na ordem inversa.

```c
/* Usando gets e putchar */
#include <stdio.h>

main(){

char sentence[80];
void reverse(char *);

printf("Digite uma linha de texto:\n");
gets(sentence);

printf("\nA linha impressa em ordem inversa e:\n");
reverse(sentence);

return 0;
}

void reverse (char *s){
if  (s[0]   == '\0')
        return;
else {
        reverse(&s[1]);
        putchar(s[0]);
}
}
```

```
Digite uma linha de texto:
Caracteres e Strings

A linha impressa em ordem inversa e:
sgnirtS e seretcaraC
```

```
Digite uma linha de texto:
orava o avaro

A linha impressa em ordem inversa e:
crava o avaro
```

Fig. 8.13 Usando **gets** e **putchar.**

```
/* Usando getchar e puts */
#include <stdio.h>

main() {

char c,  sentence[80];
int i = 0;

puts("Digite uma linha de texto:");
while (( c = getchar())!='\n')
        sentence[i++]  = c;

sentence[i]   =  '\0';  /* insere NULL no final da string */
puts("\nA linha digitada foi:");
puts(sentence);
return 0;
}
```

```
Digite uma linha de texto:
Isso e um teste.
A linha digitada foi:
Isso e um teste.
```

Fig. 8.14 Usando **getchar** e **puts.**

O programa da Fig. 8.14 usa as funções ***getchar*** e **puts** para ler os caracteres do dispositivo-padrão de entrada para o array de caracteres **sentence** e imprime o array de caracteres como uma string. A função **getchar** lê um caractere do dispositivo-padrão de entrada e retorna o caractere com um inteiro. A função **puts** apanha a string **(char *)** como argumento e imprime a string seguida de um caractere de nova linha.

O programa interrompe a entrada cie caracteres quando **getchar** ler o caractere de nova linha fornecido pelo usuário para terminar a linha de texto. Um caractere **NULL** é adicionado ao array **sentence** para que ele possa ser tratado como uma string. A função **puts** imprime a string contida em sentence**.**

```
/* Usando sprintf */
#include <stdio.h>
main(){
char s[80];
int x;
float y;

printf("Digite um valor inteiro e um valor float:\n");
scanf("%d%f",  &x,  &y);

sprintf(s,"Inteiro:%6d\nFloat:%8.2f", x, y);
printf ("%s\n%s\n",
        "A saida formatada armazenada no array s e:",  s);

return 0;
}
```

```
Digite um valor inteiro e um valor float:
298    87.375
A, saida formatada armazenada no array s e:
Inteiro: 298
Float: 87.38
```

Fig. 8.15 Usando **sprintf.**

O programa da Fig. 8.15 usa a função ***sprintf*** para imprimir dados formatados no array**s - um** array de caracteres. A função usa as mesmas especificações de conversão de **printf** (veja o Capítulo 9 para obter uma análise detalhada de todos os recursos de formatação de impressão). O programa obtém um valor **int** e um valor fl**oat** do dispositivo de entrada a serem formatados e impressos no array **s. O** array **s** é o primeiro argumento de **sprintf.** O programa da Fig. 8.16 usa a função ***sscanf*** para ler dados formatados do array de caracteres **s.** A função usa as mesmas especificações de conversão de **scanf. O** programa lê um **int** e um fl**oat do A arr**ay **s** e armazena os valores em **x** e **y,** respectivamente. Os valores de **x** e **y** são impressos. O array s é o primeiro argumento de **sscanf.**

```
/* Usando sscanf */
#include <stdio.h>

main(){

char s[] = "31298 87.375";
int x;
float y;

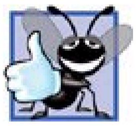
sscanf(s,"%d%f", &x, &y);
printf("%s\n%s%6d\n%s%8.3f\n",
        "Os valores armazenados no array de caracteres s sao:",
        "Inteiro:", x,   "Float:", y);
return 0;
}
```

```
Os valores armazenados no array de caracteres s sao:
inteiro: 31298
Float:     87.375
```

Fig. 8.16 Usando **sscanf.**

8.6 Funções de Manipulação de Strings da Biblioteca de Manipulação de Strings

A biblioteca de manipulação de strings fornece muitas funções úteis para manipulação de dados em strings, comparação de strings, pesquisa de caracteres e strings em outras strings, divisão de strings (separação de strings em partes lógicas) e determinação de comprimento de strings. Esta seção apresenta as funções de manipulação de strings da biblioteca de manipulação de strings. As funções estão resumidas na Fig. 8.17.



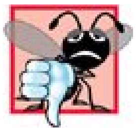
Boa prática de programação 8.5

Ao usar funções da biblioteca de manipulação de strings, inclua o arquivo de cabeçalho ***<string.h>.***

Protótipo da função	**Descrição da função**
char *strcpy(char *sl, const char *s2)	
	Copia a string s2 para o array sl. O valor de sl é retornado
char *strncpy(char *sl, const char *s2, size_t n)	
	Copia no máximo n caracteres da string s2 para o array sl. O valor de sl é retornado.
char *strcat(char *sl, const char *s2)	
	Anexa a string s2 ao array sl. O primeiro caractere de s2 é sobrescrito ao caractere NULL de terminação de sl. O valor de s1 é retornado.
char *strncat(char *sl, const char *s2, size_t n)	
	Anexa no máximo n caracteres da string s2 ao array sl. O primeiro caractere de s2 é sobrescrito ao caractere NULL de terminação de a sl. O valor de sl é retornado.

Fig. 8.17 As funções de manipulação de strings da biblioteca de manipulação de strings.

A função ***strcpy*** copia seu segundo argumento — uma string — para seu primeiro argumento — um array - de caracteres que deve ser suficientemente grande para armazenar a string e seu caractere **NULL** de terminação que também é copiado. A função ***strncpy é*** equivalente a **strcpy,** exceto que **strncpy especi**fica o número de caracteres a serem copiados da string para o array. Observe que a função **strncpy não copia** necessariamente o caractere **NULL** de terminação do segundo argumento. O caractere **NULL** de terminação escrito apenas se o número de caracteres a serem copiados for pelo menos maior em um caractere do que o comprimento da string. Por exemplo, se **"teste"** for o segundo argumento, só será escrito um caractere NULL de terminação se o terceiro argumento de st**rncpy** for pelo menos **6** (5 caracteres em **"teste" mais um c**aractere **NULL** de terminação). Se o terceiro argumento for maior do que **6,** são adicionados caracteres **NULL** ao array até que o número total de caracteres especificado pelo terceiro argumento seja escrito.



Erro comun de programação 8.6

*Não adicionar um caractere **NULL** de terminação ao primeiro argumento de um **strncpy** quando o terceiro argumento for menor ou igual ao comprimento da string no segundo argumento.*

O programa da Fig. 8.18 usa **strcpy** para copiar toda a string do array **x** para o array **y** e usa **strncpy** para copiar os **5** primeiros caracteres do array **x** para o array z. Um caractere **NULL ('\0')** é **anexado** ao array **z** porque a chamada para **strncpy** no programa não escreve um caractere **NULL** de termina**ção** (o terceiro argumento é menor do que o comprimento da string do segundo argumento).

```
/* Usando strcpy e strncpy */
#include <stdio.h>
#include <string.h>

main() {

char x[]  =  "Feliz Aniversário";
char y[20],  z [6] ;

printf("%s%s\n%s%s\n",
          "A string no array x e:   ", x,
          "A string no array y e:   ",  strcpy(y, x));

strncpy(z, x, 5); z[5]   = '\0';

printf("A string no array z e: %s\n", z);

return 0;
}
```

```
A string no array x e: Feliz Aniversário
A string no array y e: Feliz Aniversário
A string no array z e: Feliz
```

Fig. 8.18 Usando **strcpy** e **strncpy.**

A função ***strcat*** anexa seu segundo argumento — uma string — ao seu primeiro argumento — um array de de caracteres contendo uma string. O primeiro caractere do segundo argumento substitui o **NULL ('\0')** que termina a string no primeiro argumento. O programador deve se assegurar de que o array usado para armazenar a primeira string seja suficientemente grande para armazenar a primeira string, a segunda string e o caractere **NULL** de terminação (copiado da segunda string). A função **strncat** anexa **um n**úmero especificado de caracteres da segunda string à primeira string. Um caractere **NULL** de terminação é anexado automaticamente ao resultado. O programa da Fig. 8.19 demonstra as funções **strcat e strncat.**

```
/* Usando strcat e strncat */
#include <stdio.h>
#include <string.h>

main(){
char sl[20]  = "Feliz ";
char s2[]   =  "Ano Novo ";
char s3 [40] = " ";

printf("sl = %s\ns2 = %s\n",  sl, s2);
printf("strcat(sl,  s2)  = %s\n",  strcat(sl, s2));
printf("strncat(s3,  sl,  6)  = %s\n",  strncat(s3,  sl, 6));
printf("strcat(s3,  sl)  = %s\n",  strcat(s3,  sl) ) ;

return 0;
}
```

```
sl = Feliz s2 = Ano Novo
strcat(sl,  s2)  = Feliz Ano Novo
strncat(s3,  sl,  6)  = Feliz
strcat(s3,  sl)  = Feliz Feliz Ano Novo
```

Fig. 8.19 Usando **strcat** e **strncat.**

8.7 Funções de Comparação da Biblioteca de Manipulação de Strings

Esta seção apresenta as funções de comparação de strings, **strcmp** e **strncmp,** da biblioteca de manipulação de strings. Os cabeçalhos das funções e uma breve descrição de cada uma delas aparece na Fig. 8.20.

Protótipo da função	Descrição da função
int strcmp(const char *sl, const char *s2)	
	Compara a string **sl** com a string **s2.** A função retoma 0, menor do que 0 ou maior do que () se **s1** for igual a, menor do que ou maior do que s1 respectivamente.
int strncmp(const char *sl, const char *s2, size_t n)	
	Compara até **n** caracteres da string **sl** com a string **s2.** A função retorna 0, menor do que 0 ou maior do que 0 se **sl** for igual a, menor do que ou maior do que **s2,** respectivamente.

Fig. 8.20 As funções de comparação de strings da biblioteca de manipulação de string

O programa da Fig. 8.21 compara três strings usando as funções ***strcmp*** e ***strncmp. A*** função strcmp compara seu primeiro argumento string com seu segundo argumento string, caractere por caractere. A função retorna 0 se as strings forem iguais, um valor negativo se a primeira string for menor do que a segunda e um valor positivo se a primeira string for maior do que a segunda. A função **strncmp** é equivalente a **strcmp,** exceto que **strncmp** compara até um número especificado de caracteres. A função str**ncmp** não compara caracteres após um caractere **NULL** em uma string. O programa imprime o valor inteiro retornado em cada chamada da função.

```
/* Usando strcmp e strncmp */
#include <stdio.h>
#include <string.h>

main() {

char *sl = "Feliz Ano Novo";
char *s2 = "Feliz Ano Novo";
char *s3 =  "Feliz Natal";

printf ("%s%s\n%s%s\n%s%s\n\n%s%2d\ns&s%2d\n%s%2d\n\n",
        "s1 =  ",  s1,   "s2 =  ",  s2,   "s3 =  ", s3,
        "strcmp(sl,  s2)  =  ",  strcmp(sl, s2),
        "strcmp(sl,  s3)  =  ",  strcmp(sl, s3),
        "strcmp(s3,  sl)  =  ",  strcmp(s3, sl));
printf ("%s%2d\n%s%2d\n%s%2d\n",
        "strncmp(sl, s3, 6) = ", strncmp(sl, s3, 6),
        "strncmp(sl, s3, 7) = ", strncmp(sl, s3, 7),
        "strncmp(s3,  sl,  7)  =  ",  strncmp(s3,  sl, 7));

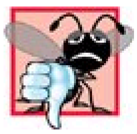
return 0;
```

```
}
```

```
sl = Feliz Ano Novo
s2 = Feliz Ano Novo
s3 = Feliz Natal
strcmp(sl, s2) =0
strcmp(sl, s3) =1
strcmp(s3,  sl)  = -1

strncmp(sl, s3, 6) = 0
strncmp(sl, s3, 7) = 1
strncmp(s3,  sl,  7)  = -1
```

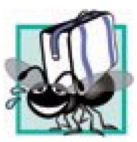
Fig. 8.21 Usando **strcmp** e **strncmp**.



Erro comun de programação 8.7

*Admitir que **strcmp** e **strncmp** retornam 1 quando seus argumentos forem iguais. Ambas as funções retornam 0(valor falso do C) em caso de igualdade. Portanto, ao examinar a igualdade de duas strings, o resultado das funções **strcmp** e **strncmp** dever ser comparado com 0 para determinar se as strings são iguais.*

Para entender exatamente o que significa uma string ser "maior do que" ou "menor do que" outra string, examine o processo de colocar em ordem alfabética uma série de sobrenomes. O leitor, indubitavelmente, colocaria "Jorge" antes de "Silva" porque a primeira letra de "Jorge" vem antes da primeira letra de "Silva" no alfabeto. Mas o alfabeto utilizado pelo computador é mais do que apenas lista de 26 letras — ele é uma lista ordenada de caracteres. Cada letra aparece em uma posição especifica dentro da lista. "z" é mais do que meramente uma letra do alfabeto; "Z" é especificamente a vigésima sexta letra do alfabeto. Como o computador sabe que uma determinada letra vem antes de outra? Todos os caracteres são apresentados no interior de um computador como códigos numéricos; quando o computador compara duas strings, na realidade ele compara os códigos numéricos dos caracteres nas strings.



Dicas de portabilidade 8.2

Os códigos numéricos usados para representar caracteres podem ser diferentes em diferentes computadores.

Procurando padronizar as representações de caracteres, a maioria dos fabricantes de computadores des**envolveu** seus equipamentos para que utilizassem um dos dois esquemas populares de codificação - ***ASCII*** ou ***EBCDIC.*** ASCII significa "American Standard Code for Information Interchange" e EBCDIC significa "Extended Binary Coded Decimal Interchange Code". Há outros esquemas, mas esses dois são os mais populares.

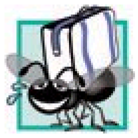
ASCII e EBCDIC são chamados ***códigos de caracteres*** ou ***conjuntos de caracteres.*** Na realidade, as manipulações de strings e caracteres envolvem a manipulação dos códigos numéricos apropriados e não dos caracteres em si. Isso explica o intercâmbio entre caracteres e inteiros pequenos em C. Como é mais significativo dizer que um código numérico é maior do que, menor do que ou igual a outro código numéri**co,** torna-se possível relacionar vários caracteres e strings entre si, fazendo referência aos códigos dos caracteres. O Apêndice C contém uma lista dos códigos de caracteres ASCII.

8.8 Funções de Pesquisa da Biblioteca de Manipulação de Strings

Esta seção apresenta as funções da biblioteca de manipulação de strings usadas para procurar caracteres e strings em outras strings. As funções estão resumidas na Fig. 8.22. Observe que as funções **strcspn e strspn** especificam o tipo de retorno **size_t. O** tipo **size_t** é um tipo definido pelo padrão como inteiro do valor retornado pelo operador **sizeof.**

Protótipo da função	Descrição da função
char *strchr(const char *s, int c)	Localiza a primeira ocorrência do caractere **c** na string **s.** Se **c** for encontrado, é retornado um ponteiro para **c.** Caso contrário, é retornado um ponteiro **NULL.**
size_t strcspn(const char *sl, const char *s2)	Determina e retorna o comprimento do segmento inicial da string **sl** consistindo nos caracteres que não estão contidos na string **s2.**
size_t strspn(const char *sl, const char *s2)	Determina e retorna o comprimento do segmento inicial da string **sl** consistindo apenas nos caracteres que estão contidos na string **s2.**
char *strpbrk(const char *sl, const char *s2)	Localiza na string **sl** a primeira ocorrência de qualquer caractere na string **s2.** Se um caractere da string **s2** for encontrado, é retornado um ponteiro para o caractere na string **sl.** Caso contrário, é retornado um ponteiro **NULL.**
char *strrchr(const char *s, int c)	Localiza a última ocorrência de **c** na string **s.** Se **c** for encontrado, é retornado um ponteiro para **c** na string **s.** Caso contrário, é retornado um ponteiro **NULL.**
char *strstr(const char *sl, const char *s2)	Localiza a primeira ocorrência da string **s2** na string **sl.** Se a string for encontrada, é retornado um ponteiro para a string em **sl.** Caso contrário, é retornado um ponteiro **NULL.**
char *strtok(char *sl, const char *s2)	Uma seqüência de chamadas a **strtok** divide a string **sl** em "tokens" ("pedaços" ou "fichas") — trechos lógicos, como palavras em uma linha de texto — separados pelos caracteres contidos na string **s2. A** primeira chamada contém **sl** como primeiro argumento, e as chamadas subseqüentes para continuar a dividir a mesma string contêm **NULL** como primeiro argumento. Cada chamada retorna ponteiro para a divisão (token) atual. Se não houver mais tokens quando a função for chamada, é retornado **NULL.**

Fig. 8.22 Funções de manipulação de strings da biblioteca de manipulação de string



Dicas de portabilidade 8.3

O tipo **size_t é u***m sinônimo dependente do sistema para o tipo* **unsigned long** *ou* **unsigned int.**

A função ***strchr*** procura pela primeira ocorrência de um caractere em uma string. Se o caracter for encontrado, **strchr** retorna um ponteiro para o caractere na string, caso contrário **strchr** retorna **NULL. O** programa da Fig. 8.23 usa **strchr** para procurar a primeira ocorrência de '**a'** e 'z' na string "I**sso e um teste".**

A função ***strcspn*** (Fig. 8.24) determina o comprimento da parte inicial da string de seu primeiro argumento que não contém quaisquer caracteres da string de seu segundo argumento. A função retorna o comprimento do segmento.

A função ***strbrk*** procura em seu primeiro argumento string pela primeira ocorrência de quais quer caracteres em seu segundo argumento string. Se for encontrado um caractere do segundo argumento, **strpbrk** retorna um ponteiro para o caractere no primeiro argumento, caso contrário **strpbrk** retorna **NULL. O** programa da Fig. 8.25 localiza em **string1** a primeira ocorrência de qualquer caracter de **string2.**

A função st**rrchr** procura em uma string pela última ocorrência do caractere especificado. Se o caractere for encontrado, **strrchr** retorna um ponteiro para o caractere na string, caso contrário **strrchr** retorna **NULL. O** programa da Fig. 8.26 procura pela última ocorrência do caractere 'z' na string **"Um zoo tem muitos animais incluindo zebras".**

A função ***strspn*** (Fig. 8.27) determina o comprimento da parte inicial da string em seu primeiro argumento que contém apenas caracteres da string em seu segundo argumento. A função retorna o comprimento do segmento.

A função ***strstr*** procura pela primeira ocorrência da string de seu segundo argumento na string de seu primeiro argumento. Se a segunda string for encontrada na primeira string, é retornado um ponteiro para o local da string no primeiro argumento. O programa da Fig. 8.28 usa **strstr** para encontrar a string **"def"** na string "a**bcdef abcdef".**

```
/* Usando strchr */
#include <stdio.h>
#include <string.h>
main(){
char *string = "Isso e um teste";
char characterl =   'm',  character2 =   ' z ' ;
if  (strchr(string,  characterl)   != NULL)
      printf("\'%c\'  foi encontrado em \"%s\".\n", characterl, string);
else
      printf("\'%c\'  nao foi encontrado em \"%s\".\n", characterl,  string);

if  (strchr(string,  character2)   != NULL)
      printf("\'%c\'  foi encontrado em \"%s\".\n", character2, string);
else
```

```
        printf("\'%c\'  nao foi encontrado em \"%s\".\n", character2, string);
return 0;
}
```

```
'm'  foi encontrado em "Isso e um teste"
'z' nao foi encontrado em "Isso e um teste".
```

Fig. 8.23 Usando **strchr**.

```
/* Usando strcspn */
#include <stdio.h>
#include <string.h>
main(){
char *stringl = "O valor e 3.14159";
char *string2 = "1234567890";
printf("%s%s\n%s%s\n\n%s\n%s%u",
        "stringl =  ",  stringl,   "string2 =  ", string2,
        "Comprimento do segmento inicial de stringl",
        "que nao contem caracteres de string2 = ",
        strcspn(stringl,  string2));
return 0;
}
```

```
stringl = O valor e 3.14159
string2 = 1234567890
Comprimento do segmento inicial de stringl
e nao contem caracteres de string2 = 10
```

Fig. 8.24 Usando **strcspn**.

```
/* Usando strpbrk */
#include <stdio.h>
#include <string.h>
main() {
char *stringl = "Isso e um teste";
char *string2 = "certo";
printf("%s\"%s\"\n'%c'%s\n\"%s\"\n", "Dos caracteres em ",
        string2, *strpbrk(stringl,  string2),
        " e o primeiro caractere a aparecer em ", stringl);

return 0;
}
```

```
Dos caracteres em "certo"
'o'  e o primeiro caractere a aparecer em
"Isso e um teste"
```

Fig. 8.25 Usando **strpbrk**.

```
/* Usando strrchr */
#include <stdio.h>
#include <string.h>
main(){
char *stringl = "Um zoo tem muitos animais incluindo zebras";
int c = 'z';
printf("%s\n%s'%c'%s\"%s\"\n",
      "O restante da stringl iniciando com a",
      "ultima ocorrência do caractere ", c,
      " e:   ",  strrchr(stringl,  c));

return 0;
}
```

```
O restante da stringl iniciando com a
ultima ocorrência do caractere  'z'  e: "zebras"
```

Fig. 8.26 Usando **strrchr**.

A função ***strtok*** é usada para dividir uma string em uma série de *tokens (pedaços).* Um token é uma seqüência de caracteres separados por *caracteres delimitadores* (normalmente espaços ou sinais de pontuação). Por exemplo, em uma linha de texto, cada palavra pode ser considerada um token e os espaços separando as palavras podem ser considerados delimitadores.

```
/* Usando strspn */
#include <stdio.h>
#include <string.h>
main(){
char *stringl =  "O valor e 3.14159";
char *string2 = "arelOov ";
printf("%s%s\n%s%s\n\n%s\n%s%u\n",
      "stringl =  ",  stringl,   "string2 = ",
      string2, "Comprimento do segmento inicial da stringl",
      "contendo apenas caracteres da string2 = ",
      strspn(stringl, string2));
return 0;
}
```

```
stringl = 0 valor e 3.14159
string2 = arelOov
Comprimento do segmento inicial da stringl
contendo apenas caracteres da string2 = 10
```

Fig. 8.27 Usando **strspn**.

São exigidas várias chamadas a **strtok** para dividir uma string em tokens (admitindo que a string contenha mais de um token). A primeira chamada a **strtok**

contém dois argumentos, uma string a ser dividida e uma string contendo os caracteres que separam os tokens. No programa da Fig. 8.29, a instrução

tokenPtr = strtok(string, " ");

atribui a **tokenPtr** um ponteiro para o token em **string. O** segundo argumento de **strtok, " ", indic**a que os tokens em **string** estão separados por espaços. A função **strtok** procura pelo primeiro caractere em **string** que não seja um caractere delimitador (espaço). Isso inicia o primeiro token. A seguir a função encontra o próximo caractere delimitador na string e o substitui por um caractere NULL**('\0').** Isso conclui o token atual. A função **strtok** salva um ponteiro para o caractere que segue o token em **string** e retorna um ponteiro para o token atual.

```
/* Usando strstr */
#include <stdio.h>
#include <string.h>
main(){

char *stringl =  "abcdefabcdef";
char *string2 = "def";

printf("%s%s\n'%s%s\n\n%s\n%s%s\n",
       "stringl =  ",  stringl,   "string2 =  ",
       string2, "O restante da stringl iniciando com a",
       "primeira ocorrência de string2 e: ",
       strstr(stringl, string2));

return 0;
}
```

```
stringl = abcdefabcdef
string2 = def
O restante da stringl iniciando com a
primeira ocorrência de string2 e: defabcdef
```

Fig. 8.28 Usando **strstr**.

```
/* Usando strtok */
#include <stdio.h>
#include <string.h>

main() {
char string[]  =  "Esta e uma frase com 7 tokens";
char *tokenPtr;
printf ("%s\n%s\n\n%s\n",
        "A string a ser dividida em tokens e:", string,
        "Os tokens sao:   ");

tokenPtr = strtok(string,   "   ");
while  (tokenPtr != NULL) {
```

```
        printf("%s\n", tokenPtr);
        tokenPtr = strtok(NULL,"   ");
   }
   return 0;
}
```

```
A string a ser dividida em tokens e:
        Esta e uma frase com 7 tokens

Os tokens sao:
Esta
e
uma
frase
com
7
tokens
```

Fig. 8.29 Usando **strtok**.

As chamadas subseqüentes que continuam a dividir st**ring** contêm **NULL** como primeiro argumento. O argumento **NULL** indica que a chamada a **strtok** deve continuar a divisão a partir do local em **string** salvo pela última chamada a **strtok.** Se não restarem tokens quando **strtok** for chamada, **strtok,** retorna **NULL. O p**rograma da Fig. 8.29 usa **strtok** para dividir a string **"Esta e uma frase com** 7 **tokens".** Cada token é impresso separadamente. Observe que **strtok** modifica a string de entrada, portanto deve ser feita uma cópia da string no caso de ela ser utilizada novamente no programa depois das chamadas a **strtok.**

8.9 Funções de Memória da Biblioteca de Manipulação de Strings

As funções da biblioteca de manipulação de strings apresentadas nesta seção facilitam a manipulação, comparação e pesquisa de blocos de memória. Essas funções podem manipular qualquer bloco de dados. A Fig. 8.30 oferece um resumo das funções de memória, da biblioteca de manipulação de strings. Nas análises das funções, "objeto" se refere a um bloco de dados.

Os ponteiros que são parâmetros dessas funções são declarados **void *.** No Capítulo 7, vimos que um ponteiro para qualquer tipo de dados pode ser atribuído diretamente a um ponteiro do tipo **void ***e um ponteiro do tipo **void *** pode ser atribuído diretamente a um ponteiro de qualquer tipo. Por esse motivo, essas funções podem receber ponteiros de quaisquer tipos de dados. Por um ponteiro do tipo **void*** não poder ser desreferenciado, cada função recebe um tamanho como argumento que especifica número de caracteres (bytes) que a função processará. Para simplificar, os exemplos desta seção manipulam arrays de caracteres (blocos de caracteres).

Protótipo da função	Descrição da função
void *memcpy(void *s1, const void *s2, size_t n)	
	Copia n caracteres do objeto apontado para s2 para o objeto apontado por s1. É retornado um ponteiro para o objeto resultante.
void *memmove (void *s1, const void *s2, size_t n)	
	Copia n caracteres do objeto apontado por s2 para o objeto apontado por s1. A cópia é realizada como se os caracteres fossem copiados primeiramente do objeto apontado por s2 para um array temporário e depois do array temporário para o objeto apontado por s1. É retornado um ponteiro para o objeto resultante.
int memcmp (const void *s1, const void *s2, size_t n)	
	Compara os n primeiros caracteres dos objetos apontados mpor s1 e s2. A função retorna 0, menor do que 0 e maior do que zero se s1 for igual a, menor do que ou maior do que s2.
void *memchr (const void *s, int c, size_t n)	
	Localiza a primeira ocorrência de c (convertido para unsigned char) nos n primeiros caracteres do objeto apontado por s. Se c fpr encontrado, é retornado um ponteiro para o objeto. Caso contrário, é retornado NULL.
void *memset (void *s, int c, size_t n)	
	Copia c(convertido para unsigned char) para os n primeiros characters do objeto apontado por s. É retornado um ponteiro para o resultado.

Fig. 8.30 As funções de memória da biblioteca de manipulação de strings.

```
/* Usando memcpy */
#include <stdio.h>
#include <string.h>
main(){
char sl[17],  s2[]  =  "Copie esta string";
memcpy(sl,  s2, 17);
printf ("%s\n%s\"%s\" \n",
        "Depois de s2 ser copiada para sl com memcpy,",
        "sl fica ", sl);
return 0;
}
```

```
Depois de s2 ser copiada para sl com memcpy,
sl fica "Copie esta string"
```

Fig. 8.31 Usando **memcpy**.

A função **mem*cpy*** copia um número especificado de caracteres do objeto apontado por seu segundo argumento para o objeto apontado por seu primeiro argumento. A função pode receber um ponteiro de qualquer tipo de objeto. O resultado dessa função é indefinido se houver superposição dos dois objetos na memória, i.e., se eles forem partes de um mesmo objeto. O programa da Fig. 8.31 usa mem**cpy para** copiar string do array s2 para o array s1**.**

```
/* Usando memmove */
#include <stdio.h>
#include <string.h>
main() {
char x[]  = "Lar Doce Lar";
printf("%s%s\n",
       "A string no array x antes de memmove e:", x);
printf("%s%s\n",
       "A string no array x depois de memmove e:",
        memmove(x,  &x[4], 8));
return 0;
}
```

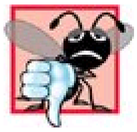
```
A string no array x antes de memmove e:
        Lar Doce Lar

A string no array x depois de memmove e:
        Doce Lar Lar
```

Fig. 8.32 Usando **memmove**.

A função mem***move,*** como a função **memcpy,** copia um número especificado de bytes do objeto apontado por seu segundo argumento para o objeto apontado por seu primeiro argumento. A cópia é realizada como se os bytes fossem copiados

primeiramente do segundo argumento para um array temporário de caracteres e depois copiados do array temporário para o primeiro argumento. Isso permite que os caracteres de uma parte de uma string sejam copiados em outra parte da mesma string.



Erro comun de programação 8.8

As funções de manipulação de strings diferentes de ***memmove*** *que copiam caracteres apresentam resultados incertos quando a cópia ocorre entre partes de uma mesma string.*

O programa da Fig. 8.32 **usa memmove** para copiar os **10** últimos, bytes do array **x** nos **10** primeiros bytes do array **x.**

A função m***emcmp*** (Fig. 8.33) compara o número especificado de caracteres de seu primeiro argumento com os caracteres correspondentes de seu segundo argumento. A função retorna um valor maior do que 0 se o primeiro argumento for maior do que o segundo, retorna 0 se os argumentos forem iguais e retorna um valor menor do que 0 se o primeiro argumento for menor do que o segundo.

A função ***memchr*** procura pela primeira ocorrência de um byte, representado como **unsigned char** no número especificado de bytes de um objeto. Se o byte for encontrado, a função retorna um ponteiro para o byte no objeto, caso contrário, a função retorna um ponteiro **NULL. O** programa da Fig. 8.34 procura pelo caractere (byte) 'r' na string " I**sto e uma string".**

A função mem***set*** copia o valor do byte em seu segundo argumento para um número especificado de bytes do objeto apontado pelo seu primeiro argumento. O programa da Fig. 8.35 **usa memset para** copiar 'b' nos 7 primeiros bytes de **string1.**

8.10 Outras Funções da Biblioteca de Manipulação de Strings

As duas funções restantes da biblioteca de manipulação de strings são **strerror** e **strlen.** A 8.36 resume as funções **strerror** e **strlen.**

```
/* Usando memcmp */
#include <stdio.h>
#include <string.h>
main(){
char sl[] = "ABCDEFG", s2[] = "ABCDXYZ";
printf("%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n",
      "sl =  ",  sl,   "s2 =  ", s2,
      "memcmp(sl,  s2, 4)  = ", memcmp(sl, s2, 4),
      "memcmp(sl,  s2,  1)  =  ", memcmp(sl,  s2, 7),
      "memcmp(s2,  sl,  7)  =  ", memcmp(s2,  sl, 7));
return 0;
}
```

```
sl  = ABCDEFG
S2  = ABCDXYZ
semcmp(sl, s2, 4) = 0
memcmp(sl,  s2,  7)  = -1
memcmp(s2,  sl,  7)  = 1
```

Fig. 8.33 Usando **memcmp**.

```
/* Usando memchr */
#include <stdio.h>
#include <string.h>
main(){
char *s =  "Isso e uma string";
printf("%s\'%c\'%s\"%s\"\n",
      "0 restante de s depois do caractere ", 'r',
      " ser encontrado e ", memchr(s,   'r', 17));
return 0;
}
```

```
0 restante de s depois do caractere  'r'  ser encontrado e "ring"
```

Fig. 8.34 Usando **memchr**.

```
/* Usando memset */
#include <stdio.h>
#include <string.h>
main(){
char stringl[15]   =  "BBBBBBBBBBBBBB";
printf("stringl = %s\n",  stringl);
printf("stringl apos memset = %s\n",
memset(stringl,   'b', 7));
return 0;
}
```

```
Stringl = BBBBBBBBBBBBBB
stringl apos memset = bbbbbbbBBBBBBB
```

Fig. 8.35 Usando **memset**.

Protótipo da função	**Descrição da função**
char *strerror(int errornum)	Mapeia errornum em uma string complete de texto de uma forma dependente do sistema. É retornado um ponteiro para a string.
size_t strlen(const char *s)	Determina o comprimento da string s. É retornado o número de caracteres que antecedem o caractere de terminação NULL.

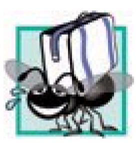
Fig. 8.36 As funções de manipulação de strings da biblioteca de manipulação de string.

```
/* Usando strerror */
#include <stdio.h>
#include <string.h>
main(){
printf("%s\n",  strerror(2));
return 0;
}
```

```
Error 2
```

Fig. 8.37 Usando **strerror**.

A função ***strerror*** utiliza um número de erro e cria uma string de mensagem de erro. É retornado um ponteiro para a string. O programa da Fig. 8.37 demonstra **strerror.**



Dicas de portabilidade 8.4

*A mensagem gerada por **strerror** depende do sistema utilizado.*

A função **strl*en*** utiliza um argumento e retorna o número de caracteres de uma string — o caractere **NULL** de terminação não está incluído no comprimento. O programa da Fig. 8.38 demonstra a função **strlen.**

```
/* Usando strlen */
#include <stdio.h>
#include <string.h>

main(){

char *stringl =  "abcdefghijklmnopqrstuvwxyz";
char *string2 = "four";
char *string3 = "Boston"^

printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
      "0 comprimento de ", stringl, " e ", strlen(stringl),
      "O comprimento de ", string2, " e ", strlen(string2),
      "0 comprimento de ",  string3,   " e ",  strlen(string3));

return 0;
}
```

```
O comprimento de "abcdefghijklmnopqrstuvwyz" e 26
O comprimento de "four" e 4
O comprimento de "Boston" e 6
```

Fig. 8.38 Usando **strlen.**

Resumo

• A função **islower** determina se seu argumento é uma letra minúscula **(a — z).**
• A função is**upper** determina se seu argumento é uma letra maiúscula **(A-Z).**
• A função is**digit** determina se seu argumento é um dígito **(0-9).**
• A função **isalpha** determina se seu argumento é uma letra maiúscula **(A-Z)** ou minúscula (a — z)
• A função is**alnum** determina se seu argumento é uma letra maiúscula **(A- Z),** uma letra minúscula **(a — z)** ou um dígito **(0-9).**
• A função **isxdigit** determina se seu argumento é um dígito hexadecimal **(A-F, a — f, 0-9).** A função **toupper** converte uma letra minúscula em maiúscula e retorna a letra maiúscula.
• A função **tolower** converte uma letra maiúscula em minúscula e retorna a letra minúscula. A função **isspace** determina se seu argumento é um dos seguintes caracteres de espaço em branco: ' ' (espaço), ' **\f', '\n', '\r', '\t' ou '\v'.**
• A função **iscntrl** determina se seu argumento é um dos seguintes caracteres de controle: '**\t', '\v',** '\v', '\a', '\f', '\a', '\b', '**\r''** ou '\n'.
• A função is**punct** determina se seu argumento é um caractere imprimivel diferente de um espaço, um dígito ou uma letra.
• A função **isprint** determina se seu argumento é qualquer caractere imprimivel incluindo o caractere de espaço.
• A função **isgraph** determina se seu argumento é qualquer caractere imprimivel diferente do caractere de espaço.
• A função **atof** converte seu argumento — uma string iniciando com uma série de dígitos que representam um número de ponto flutuante — em um valor **double.**
• A função **atoi** converte seu argumento — uma string iniciando com uma série de dígitos que representam um número inteiro — em um valor **int.**
• A função **atoi** converte seu argumento — uma string iniciando com uma série de dígitos que representam um número inteiro longo — em um valor **long.**
• A função **strtod** converte uma seqüência de caracteres representando um valor de ponto flutuante em **double.** A função recebe dois argumentos — uma string **(char *)** e um ponteiro para **char *.** A string contém a seqüência de caracteres a ser convertida, e ao ponteiro para **char** * é atribuído **o** restante da string após a conversão.
• A função **strtol** converte uma seqüência de caracteres representando um valor inteiro em **long.** A função recebe três argumentos — uma string **(char *),** um ponteiro para **char** * e um inteiro. A string contém a seqüência de caracteres a ser convertida, ao ponteiro para **char** * é atribuído o restante da string após a conversão e o inteiro especifica a base do valor que está sendo convertido.
• A função **strtoul** converte uma seqüência de caracteres representando um inteiro em **unsigned long.** A função recebe três argumentos — uma string **(char *),** um ponteiro para **char** * e um inteiro. A string contém a seqüência de caracteres a ser convertida, ao ponteiro para **char** * é atribuído o restante da string após a conversão e o inteiro especifica a base do valor que está sendo convertido.
• A função **gets** lê os caracteres do dispositivo-padrão de entrada (teclado) até que um caractere de nova linha ou o indicador de fim de arquivo seja encontrado. O argumento para **gets** é um array do tipo **char.** Um caractere **NULL ('\0')** é adicionado ao array após o término da leitura.
• A função **putchar** imprime seu argumento de caracteres.

• A função **getchar** lê um único caractere do dispositivo-padrão de entrada e retorna o caractere com um inteiro. Se o indicador de final de arquivo for encontrado, **getchar** retorna **EOF.**

• A função **puts** recebe uma string **(char *)** como argumento e imprime a string seguida de um caractere de nova linha.

• A função **sprintf** usa as mesmas especificações de conversão que **printf** para imprimir dados formatados em um array do tipo **char.**

• A função **sscanf** usa as mesmas especificações de conversão que a função **scanf** para ler dados formatados de uma string.

• A função **strcpy** copia seu segundo argumento — uma string — para seu primeiro argumento — um caractere. O programador deve assegurar-se de que o array é suficientemente grande para armazenar a string e seu caractere **NULL** de terminação.

• A função **strncpy** é equivalente a **strcpy,** exceto que uma chamada a **strncpy** especifica-se número de caracteres a serem copiados da string para o array. O caractere **NULL** de terminação será copiado se o número de caracteres a serem copiados for uma unidade maior do que o comprimento da string.

• A função **strcat** anexa seu segundo argumento — incluindo o caractere **NULL** de terminação — ao seu primeiro argumento. O primeiro caractere da segunda string substitui o caractere NULL da primeira string. O programador deve assegurar-se de que o array usado para armazenar a primeira string é suficientemente grande para armazenar tanto a primeira string como a segunda.

• A função **strncat** anexa um número especificado de caracteres da segunda string à primeira string. O caractere **NULL** de terminação é anexado ao resultado.

• A função **strcmp** compara a string de seu primeiro argumento com a string de seu segundo argumento, caractere por caractere. A função retorna 0 se as strings forem iguais, retoma um valor negativo se a primeira string for menor do que a segunda e retorna um valor positivo se a primeira string for maior do que a segunda.

• A função **strncmp** é equivalente a **strcmp,** exceto que **strncmp** compara um número especificado de caracteres. Se o número de caracteres em uma das strings for menor do que o número de caracteres especificado, **strncmp** compara os caracteres até o caractere **NULL** da string menor se encontrado.

• A função **strchr** procura pela primeira ocorrência de um caractere em uma string. Se o caractere for encontrado, **strchr** retorna um ponteiro para o caractere na string, caso contrário **strchr re**torna **NULL.**

• A função **strcspn** determina o comprimento da parte inicial da string em seu primeiro argumento que não contém qualquer um dos caracteres da string que se encontra em seu segundo argumento, a função retorna comprimento do segmento.

• A função **strpbrk** procura na string de seu primeiro argumento pela primeira ocorrência de qualquer caractere em seu segundo argumento. Se um caractere do segundo argumento for encontrado **strpbrk** retorna um ponteiro para o caractere, caso contrário retorna **NULL.**

• A função **strrchr** procura pela última ocorrência de um caractere em uma string. Se o caracter for encontrado, **strrchr** retorna um ponteiro para o caractere na string, caso contrário strrchr retorna **NULL.**

• A função **strspn** determina o comprimento da parte inicial da string em seu primeiro argumento que contém apenas caracteres da string no segundo argumento. A função retorna o comprimento de segmento.

• A função **strstr** procura pela primeira ocorrência da string do segundo argumento em seu primeiro argumento. Se a segunda string for encontrada no primeiro argumento, é retornado um ponteiro para a localização da string no primeiro argumento.

• Uma série de chamadas a **strtok** divide a string **s1** em partes (tokens) que são separadas pelo caracteres contidos na string s2**.** A primeira chamada contém s1 como primeiro argumento, e as chamadas subseqüentes para continuar a dividir a mesma string contêm **NULL** como primeiro argumento

• Cada chamada retorna um ponteiro para o token atual. Se não houver mais tokens quando a função for chamada, um ponteiro **NULL é** retornado.

• ***A*** função **memcpy** copia um número especificado de caracteres do objeto para o qual seu segundo argumento aponta, no objeto para o qual seu primeiro argumento aponta. **A** função pode receber um ponteiro para qualquer tipo de objeto. Os ponteiros são recebidos por **memcpy** como ponteiros **void e** convertidos para ponteiros **char** para serem usados na função.

• **A** função **memcpy** manipula os bytes do objeto como caracteres.

• A função **memmove** copia um número especificado de bytes do objeto para o qual seu segundo argumento aponta, no objeto para o qual seu primeiro argumento aponta. **A** cópia **é** realizada como se os bytes fossem copiados do segundo argumento para um array temporário de caracteres e depois copiados do array temporário de caracteres para o primeiro argumento.

• A função **memcmp** compara o número especificado de caracteres de seu primeiro e de seu segundo argumentos.

• A ***fu***nção **memchr** procura pela primeira ocorrência de um byte, representado como **unsigned char, no** número especificado de bytes de um objeto. Se o byte for encontrado, **é** retornado um ponteiro para ele, caso contrário **é** retornado um ponteiro **NULL.**

• A função m**emset** copia seu segundo argumento, tratado como **unsigned char,** para um número especificado de bytes do objeto apontado por seu primeiro argumento.

• A função **strerror m**apeia um número inteiro de erro em uma string completa de texto de uma maneira que depende do sistema. É retornado um ponteiro para a string.

• A função **strlen** recebe uma string como argumento e retorna o número de caracteres de uma string — o caractere **NULL** de terminação não **é** incluído no comprimento da string.

Terminologia

anexando strings a outras strings
ASCII
atof
atoi
atol
biblioteca geral de utilitários
caractere de controle
caractere imprimível
caracteres de espaço em branco
código de caracteres
comparando strings
comprimento de uma string
concatenação de strings
conjunto de caracteres
constante de caracteres
constante de strings
copiando strings ctype.h
delimitador dígitos hexadecimais
EBCDIC
funções de comparação de strings
funções de conversão de strings
getchar
gets
isa1num
isalpha
iscntr
islower
isprint
ispunct
isspace
isupper
isxdigit
literal
memchr
memcmp
memcpy
memmove
memset
processamento de strings
processamento de textos
putchar
puts
representação de código numérico de um caractere
sprintf
sscanf
stdio.h
stdlib.h
strcat
strchr
strcmp
strcpy
strcspn
strerror
string
string de busca
string.h
string literal
strings de divisão
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtok strtol
strtoul
token
tolower
toupper

Erros Comuns de Programação

8.1 Não alocar espaço suficiente em um array de caracteres para armazenar o caractere **NULL** que termina uma string.

8.2 Imprimir uma "string" que não contém um caractere **NULL** de terminação.

8.3 Processar um caractere simples como uma string. Uma string é um ponteiro — provavelmente um inteiro grande considerável. Entretanto, um caractere é um inteiro pequeno (valores ASCII variando de 0a 255. Em muitos sistemas isso causa um erro porque os endereços pequenos da memória são reservados para finalidades especiais como handlers de interrupção do sistema operacional — dessa forma, ocorrem "violações de acesso".

8.4 Passar um caractere como argumento para uma função quando uma string é esperada.

8.5 Passar uma string como argumento para uma função quando um caractere é esperado.

8.6 Não adicionar um caractere **NULL** de terminação ao primeiro argumento de um **strncpy** quando o terceiro argumento for menor ou igual ao comprimento da string no segundo argumento.

8.7 Admitir que **strcmp** e **strncmp** retornam 1 quando seus argumentos forem iguais. Ambas as funções retornam 0 (valor falso do C) **cm** caso de igualdade. Portanto, ao examinar a igualdade de duas strings, o resultado das funções **strcmp** e **strncmp** devem ser comparados com 0 para determinar se as strings são iguais

8.8 As funções **de** manipulação **de** strings diferentes **de memmove** que copiam caracteres apresentam resultados incertos quando a cópia ocorre entre partes **de** uma mesma string.

Práticas Recomendáveis de Programação

8.1 Ao armazenar uma string **de** caracteres **cm** um array, certifique-se **de** que o array é suficientemente grande para conter a maior string que será armazenada. A linguagem C permite que sejam armazenadas strings de qualquer comprimento. Se uma string for maior **do** que o array **de** caracteres **no** qual está sendo armazenado, os caracteres além **do** final **do** array irão sobrescrever os dados **da** memória após o array.

8.2 Ao usar funções **da** biblioteca **de** manipulação **de** caracteres, inclua o arquivo **de** cabeçalho **<ctype.h>**

8.3 Ao usar funções **da** biblioteca geral **de** utilitários, inclua o arquivo **de** cabeçalho **<stdlib.h>.**

8.4 Ao usar funções **da** biblioteca-padrão **de** entrada/saída, inclua o arquivo **de** cabeçalho <s**tdio.h>.**

8.5 Ao usar funções **da** biblioteca **de** manipulação **de** strings, inclua o arquivo **de** cabeçalho **<string.h>**

Dicas de Portabilidade

8.1 Quando uma variável **do** tipo **char** * for inicializada com uma string literal, alguns compiladores podem colocar a string **em** um local **da** memória onde ela não pode ser modificada. Se for necessário **modificar uma** string, ela deverá ser armazenada em um array **de** caracteres para que seja assegurada a **possibilidade de** modificá-la em todos os sistemas.

8.2 Os códigos numéricos usados para representar caracteres podem ser diferentes **em** diferentes c**omputadores**

8.3 O tipo **size_t** é um sinônimo dependente **do** sistema para o tipo **unsigned long** ou **unsigned.**

8.4 A mensagem gerada por **strerror** depende **do** sistema utilizado.

Exercícios de Revisão

8.1 Escreva uma instrução simples que realize cada um dos seguintes pedidos. Admita que as variáveis c (que armazena um caractere), **x, y** e **z** são **do** tipo **int,** as variáveis **d. e** e **f** são **do** tipo fl**oat,** a variavel ptr é do tipo **char** * e os arrays **s1[100] e s2[100]** são **do** tipo **char.**

a) Converta o caractere armazenado **na** variável **c** em uma letra maiúscula. Atribua o resultado à variavel c.

b) Determine se o valor **da** variável **c** é um dígito. Use o operador condicional **da** forma apresentada nas Figs. 8.2, 8.3 e 8.4 para imprimir " **e um "** ou " **nao e um "** quando o resultado for apresentada

c) Converta a string **"1234567"** em **long** e imprima o valor.

d) Determine se o valor **da** variável **c** é um caractere **de** controle. Use o operador condicional para imprimir " **e um "** ou " **nao e um "** quando o resultado for apresentado.

e) Leia uma linha de texto a partir do teclado e armazene-a no array s1**.** Não use **scanf.**

f) Imprima a linha de texto armazenada no array **s1.** Não use **printf.**

g) Atribua a **ptr** a localização da última ocorrência de **c** em **s1.**

h) Imprima o valor da variável **c.** Não use **printf.**

i) Converta a string "8.6**3582 "** em **double** e imprima o valor.

j) Determine se o valor de **c** é uma letra. Use o operador condicional para imprimir " **e uma "** ou " **nao e uma "** quando o resultado for apresentado. k) Leia um caractere do teclado e armazene-o na variável **c.**

l) Atribua a **ptr** a localização da primeira ocorrência de **s2** em s1**.**

m) Determine se o valor da variável **c** é de um caractere imprimível. Use o operador condicional para imprimir " **e um "** ou " **nao e um "** quando o resultado for apresentado.

n) Leia três valores fl**oat** da string "1.27 **10.3   9.342"** nas variáveis **d, e** e f.

o) Copie no array **s1** a string armazenada no array **s2.**

p) Atribua a **ptr** a localização da primeira ocorrência de qualquer caractere de **s1** em **s2.**

q) Compare a string em **s1** com a string em **s2.** Imprima o resultado.

r) Atribua a **ptr** a localização da primeira ocorrência de **c** em **s1.**

s) Use **sprintf** para imprimir os valores das variáveis inteiras **x, y** e **z** no array **s1.** Cada valor deve ser impresso com um comprimento de campo igual a 7.

t) Anexe 10 caracteres da string em **s2** à string em **s1.**

u) Determine o comprimento da string em s1**.** Imprima o resultado.

v) Converta a string " **- 21"** em **int** e imprima o valor.

w) Atribua a **ptr** a localização do primeiro token em **s2.** Os tokens em **s2** são separados por vírgulas (,).

8.2 Mostre dois métodos diferentes de inicializar o array de caracteres **vogal** com a string de vogais "A**EIOU".**

8.3 O que será impresso quando cada uma das seguintes instruções em C forem executadas? Se a instrução possuir um erro, descreva-o e indique como corrigi-lo. Admita as seguintes declarações de variáveis:

char s1[50]   = "jack", s2[50] = "jill", s3[50], *sptr;

a) **print("%c%s", toupper(s1[0]), &s1[1]);**

b) **printf(*"%s*", strcpy(s3, s2));**
c) **printf("%s", strcat(strcat(strcpy(s3, s1),   " e "), s2));**
d) **printf("%u", strlen(s1) + strlen(s2));**
e) **printf("%u", strlen(s3));**

8.4 Encontre o erro em cada um dos seguintes segmentos de programa e explique como corrigi-lo:

a) char s [10] ;
strncpy(s,   "hello", 5); printf("%s\n", s);
b) **printf (*"%s* ",   ' a ') ;**
c) **char s[12];**
strcpy(s,   "Bem-vindo ao Lar");
d) **if (strcmp(string1, string2)) printf ("As strings sao iguais\n");**

Respostas dos Exercícios de Revisão

8.1 a) **c = toupper(c);**
b) **printf("'%c'%sdigito\n", c, isdigit(c) ? " e um " : " nao e um ");**
c) **printf("%1d\n", atol("1234567"));**
d) **printf("'%c'%scaractere de controle\n", c, iscntrl(c) ? " e um " : " nao e um ");**
e) **gets (s1) ;**
f) puts (s1);
g) **ptr = strrchr(s1, c);**
h) **putchar(c);**
1) **printf ("%6f\.n", atof (" 8 . 63582")) ;**
j) printf("'%c'%sletra\n", c, isalpha(c) ? " e uma " : " nao e uma ");
k) **c = getchar ();**
1) **ptr = strstr(s1, s2);**
m)**printf("'%c'%scaractere imprimivel\n",**
c, isprint(c) ? " e um " : " nao e um "); n) **sscanf("1.27 10.3 9.432", "%f%f%f", &d, &e, &f);** o) **strcpy(s1, s2);**
p) **ptr = strpbrk(s1, s2);**
q) **printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2));**
r) **ptr = strchr(s1, c) ;**
s) **sprintf(s1, "%7d%7d%7d", s, y, z);**
t) **strncat(s1, s2, 10);**
u) **printf("strlen(s1) = %u\n", strlen(s1));** v) **printf("%d\n", atoi("-21"));** w) **ptr = strtok(s2, ",");**

8.2 **char vogal[] = "AEIOU";**
char vogal[] = { 'A', 'E', 'I', 'O', 'U', ' \ 0 ' } ;

8.3 a) **Jack**
b) j**ill**
c) **jack e jill**
d) **8**
e) **13**

8.4 a) Erro: A função **strncpy** não escreve um caractere **NULL** de terminação no array **s** porque seu terceiro argumento é igual ao comprimento da string **"hello".**
Correção: Faça com que o terceiro argumento de **strncpy** seja **6** ou atribua ' \ **0** ' a **s[5].**
b) Erro: Tentar imprimir uma constante de caracteres como uma string.
Correção: Use %*c* para enviar o caractere para o dispositivo de saída ou substitua '**a**' por "a".
c) Erro: O array de caracteres **s** não é suficientemente grande para armazenar o caractere **NULL** de ter**minaç**ão.
Correção: Declare o array com mais elementos.
d) Erro: A função **strcmp** retornará 0 se as strings forem iguais, portanto a condição na estrutura if será falsa e **printf** não será executado.
Correção: Compare o resultado de **strcmp** com **0** na condição.

Exercícios

8.5 Escreva um programa que receba um caractere do teclado e verifique-o com cada uma das funções manipulação de caracteres. O programa deve imprimir o valor retornado por cada função.

8.6 Escreva um programa que coloque uma linha de texto no array dc caracteres **s [100]** utilizando a função **gets.** Envie a linha para o dispositivo de saída em letras maiúsculas e em letras minúsculas.

8.7 Escreva um programa que receba 4 strings que representem inteiros, converta as strings em inteiros, some os valores e imprima a soma dos 4 valores.

8.8 Escreva um programa que receba 4 strings que representem 4 valores de ponto flutuante, converta as strings em valores **double,** some os valores e imprima a soma dos quatro valores.

8.9 Escreva um programa que use a função **strcmp** para comparar duas strings fornecidas pelo usuário. O programa deve indicar se a primeira string é menor, igual ou maior do que a segunda.

8.10 Escreva um programa que use a função **strncmp** para comparar duas strings fornecidas pelo usuário do programa deve receber o número de caracteres a serem comparados. O programa deve indicar se a primeira string é menor, igual ou maior do que a segunda.

8.11 Escreva um programa que use a geração de números aleatórios para criar frases. O programa deve usar quatro arrays de ponteiros **char** denominados **artigo, substantivo, verbo** e **preposição.** O programa deve criar uma frase selecionando uma palavra aleatoriamente de cada array na seguinte ordem: artigo, **substantivo, verbo, preposição, artigo** e **substantivo.** A medida que cada palavra for selecionada, ela deverá ser concatenada às palavras anteriores em um array que seja suficientemente grande para conter a frase inteira. As palavras devem ser separadas pofespaços. Quando a frase final for enviada para o dispositivo de saída, ela deve iniciar com uma letra maiúscula e terminar com um ponto. O programa deve gerar 20 de tais frases.

Os arrays devem ser preenchidos como se segue: o array **artigo** deve conter **"o", "um", "algum", "todo"** e **"qualquer";** o array **substantivo** deve conter **"menino", "homem", "cachorro", "carro", "gato"**; o array **verbo** deve conter **"passou", "pulou", "correu", "saltos", "andou"**; o array **preposição** deve conter **"sobre", "sob", "ante", "ate"** e **"com".**

I

Depois de o programa anterior estar escrito e funcionando, modifique-o para produzir uma pequena história consistindo em várias frases. (Imagine a possibilidade de se obter um gerador aleatório de dissertações escolares!)

8.12 *(Limericks)* A palavra inglesa *Umerick* designa um poema humorístico de cinco linhas nas quais a **primei**ra e a segunda linhas rimam com a quinta, e a terceira linha rima com a quarta. Usando técnicas similares ás desenvolvidas no Exercício 8.10, escreva um programa em C que produza limericks aleatórios. Utilizar esse programa para que produza bons limericks é um problema complexo, mas o resultado compensará o esforço!

8.13 Escreva um programa que codifique, no que será chamado aqui de "Latim pobre", frases da língua portuguesa. Usa-se o termo "Latim pobre" baseado em uma codificação de frases em inglês, chamada "pig Latin", utilizada freqüentemente para diversão. Existem muitas variações para formar frases inglesas em pig Latin. Para simplificar, use o seguinte algoritmo:

Para formar uma frase em Latim pobre a partir de uma frase em português, divida a frase em palavras (tokens) com a função **strtok.** Para traduzir cada palavra no idioma português em uma palavra em Latim pobre, coloque a primeira letra da palavra em português no final da mesma e adicione as letras "**ai"** (para codificar em pig Latin, adiciona-se "**ay"** ao final das palavras em inglês). Assim, a palavra "**salto"** torna-se "**altosai",** a palavra **"um"** torna-se "m**uai "** e a palavra **"computador"** torna-se " **omputadorcai".** Os espaços em branco entre palavras permanecem espaços em branco. Admita o seguinte: A frase em português consiste em palavras separadas por espaços em branco, não há sinais de pontuação e todas as palavras possuem duas ou mais letras. A função **imprimePalavraLatim** deve exibir cada palavra. Sugestão: Sempre que um token for encontrado em uma chamada a **strtok,** passe o ponteiro do token para a função **imprimePalavraLatim** e imprima a palavra em Latim pobre.

8.14 Escreva um programa que receba um número de telefone como uma string na forma **(555) 555-5555. O** programa deve usar a função **strtok** para extrair o código de área como um token, os três primeiros dígitos do número de telefone como outro token e os últimos quatro dígitos do número de telefone como mais outro token. Os sete dígitos do número de telefone devem ser concatenados e formar uma única string. O programa deve converter a string do código de área em **int** e converter a string do número de telefone em **long.** Tanto o código de área como o número de telefone devem ser impressos.

8.15 Escreva um programa que receba uma linha de texto, divida a linha por meio da função **strtok** e imprima o resultado na ordem inversa.

8.16 Escreva um programa que receba do teclado uma linha de texto e uma string. Usando a função **strstr,** localize a primeira ocorrência da string de busca na linha de texto e atribua a localização daquela string à variável **buscaPtr** do tipo **char *.** Se a string de busca for encontrada, imprima o restante da linha de texto iniciando com a string de busca. A seguir, use novamente **strstr** para localizar a próxima ocorrência da string de busca na linha de texto. Se for encontrada uma segunda ocorrência, imprima o restante da linha de texto iniciando com a segunda ocorrência. Sugestão: A segunda chamada a **strstr** deve conter **buscaPtr +** 1 como primeiro argumento.

8.17 Escreva um programa baseado no programa do Exercício 8.16 que receba várias linhas de texto e uma string de busca, e use a função **strstr** para determinar o total de ocorrências da string na linha de texto. Imprima o resultado.

8.18 Escreva um programa que receba várias linhas de texto e um caractere de busca, e use a função **strchr** para determinar o total de ocorrências do caractere nas linhas de texto.

8.19 Escreva um programa baseado no Exercício 8.18 que receba várias linhas de texto e use a função **strchr** para determinar o total de ocorrências de cada letra do alfabeto nas linhas de texto. As letras maiúsculas e minúsculas devem ser contadas em conjunto. Armazene o total de cada letra em um array e imprima os valores em um formato de tabela depois de os

totais terem sido determinados.

8.20 Escreva um programa que receba várias linhas de texto e use **strtok** para contar o número total de palavras. Admita que as palavras estão separadas por espaços em branco ou caracteres de nova linha.

8.21 Use as funções de comparação de strings analisadas na Seção 8.6 e as técnicas de classificar arrays desenvolvidas no Capítulo 6 para escrever um programa que coloque uma lista de strings em ordem alfabética. Use os nomes de 10 ou 15 cidades de sua região como dados para seu programa.

8.22 A tabela do Apêndice C mostra as representações dos códigos numéricos dos caracteres do conjunto ASCII. Estude essa tabela e depois diga se cada uma das afirmações a seguir é verdadeira ou falsa.

a) A letra "A" vem antes da letra "B".

b) O dígito " **9 "** vem antes do dígito " **0 ".**

c) Os símbolos usados normalmente para adição, subtração, multiplicação e divisão vem antes de qualquer um dos dígitos.

d) Os dígitos vem antes das letras.

e) Se um programa de ordenação colocar strings na ordem ascendente, o símbolo do parêntese direito virá antes do símbolo do parêntese esquerdo.

8.23 Escreva um programa que leia uma série de strings e imprima apenas as strings que começarem com a letra "b".

8.24 Escreva um programa que leia uma série de strings e imprima apenas as strings que terminarem com as letras

8.25 Escreva um programa que receba um código ASCII e imprima o caractere correspondente. Modifique esse programa para que ele gere todos os códigos possíveis de três dígitos no intervalo de 000 a 255 e tente imprimir os caracteres correspondentes. O que acontece quando esse programa é executado?

8.26 Usando a tabela de caracteres ASCII do Apêndice C como guia, escreva suas próprias versões das funções de manipulação de caracteres da Fig. 8.1

8.27 Escreva suas próprias versões das funções da Fig. 8.5 para converter strings em números.

8.28 Escreva duas versões de cada uma das funções de cópia de strings e de concatenação de strings da Fig 8.17, na primeira versão deve usar subscritos de arrays e a segunda versão deve usar ponteiros e aritmética de ponteiros,

8.29 Escreva suas próprias versões das funções **getchar, gets, putchar** e **puts** descritas na Fig. 8.12.

8.30 Escreva duas versões de cada uma das funções de comparação de strings da Fig. 8.20. A primeira versão deve usar subscritos de arrays e a segunda versão deve usar ponteiros e aritmética de ponteiros.

8.31 Escreva suas próprias versões das funções da Fig. 8.22 para pesquisar strings.

8.32 Escreva suas próprias versões das funções da Fig. 8.30 para manipular blocos de memória.

8.33 Escreva duas versões da função **strlen** na Fig. 8.36. A primeira versão deve usar subscritos de arrays e a segunda versão deve usar ponteiros e aritmética de ponteiros.

Seção Especial: Um Compêndio dos Exercícios Mais Avançados de Manipulação de Strings

Os exercícios anteriores estão voltados para textos e destinam-se a avaliar a compreensão que o leitor possuir a respeito dos conceitos fundamentais de manipulação de strings. Esta seção inclui um conjunto de problemas médios e avançados. O leitor deve achar que esses problemas são complexos, porém agradáveis. A dificuldade dos problemas pode variar consideravelmente. Alguns exigem uma ou duas horas para escrever e implementar o programa. Outros são úteis para tarefas práticas que podem exigir duas ou três semanas de estudos e implementação. Alguns são projetos escolares complexos.

8.34 *(Análise de Textos)* A disponibilidade de computadores com recursos de manipulação de strings resultou em alguns métodos muito interessantes para analisar as obras de grandes autores. Foi dedicada muita atenção para determinar se William Shakespeare realmente existiu. Alguns especialistas acreditam que há provas substanciais indicando que Christopher Marlowe realmente escreveu as obras-primas atribuídas a Shakespear. Os pesquisadores usaram computadores para encontrar similaridades entre as obras daqueles dois autores. Este exercício examina três métodos de analisar textos com um computador.

a) Escreva um programa que leia várias linhas de texto e imprima uma tabela indicando o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

To be, or not to be: that is the question:(Ser *ou não ser: eis a questão:)*

contem um "a", dois "b", nenhum "c" etc.

b) Escreva um programa que leia várias linhas dc texto e imprima uma tabela indicando o número de palavras de uma letra, palavras de duas letras, palavras de três letras etc. que aparecem no texto. Por exemplo, a frase

Whether 'tis nobler in the mind to suffer *(Se é mais nobre sofrer mentalmente)*
contém

Comprimento da palavra	Ocorrências
1	0
2	2
3	2
4	2 (incluindo ‘tis)
5	0
6	2
7	1

c) Escreva um programa que leia várias linhas de texto e imprima uma tabela indicando o número de ocorrências de cada palavra diferente no texto. A primeira versão de seu programa deve incluir as palavras ou tabela na mesma ordem em que aparecem no texto. Deve ser experimentada uma saída mais interresantes (e útil) na qual as palavras são ordenadas alfabeticamente. Por exemplo, as linhas

To be, or not to be: that is the question: Whether 'tis nobler in the mind to

suffer

contém as palavras "to" três vezes, a palavra "be" duas vezes, a palavra "or" uma vez etc.

8.35 *(Processamento de Textos) O* tratamento detalhado da manipulação de strings neste texto é, em grande parte, conseqüência do enorme crescimento do processamento de textos nos últimos anos. Uma função importante dos sistemas de processamento de textos é a *justificação de tipos* — o alinhamento das palavras nas margens direita e esquerda de uma página. Isso gera um documento com aparência profissional que parece gerado em tipografia em vez de em uma máquina de escrever. A justificação de tipos pode ser realizada em sistemas computacionais inserindo um ou mais caracteres em branco entre cada uma das palavras em uma linha, de modo que a palavra situada na extremidade direita fique alinhada com a margem direita. Escreva um programa que leia várias linhas de texto e imprima esse texto no formato justificado. Admita que o texto deve ser impresso em um papel com largura de 8,5 polegadas (21,59 cm, a largura de um papel tipo Carta) e que devem ser permitidas margens de uma polegada (2,54 cm) nos lados esquerdo e direito da página impressa. Suponha que o computador imprime 10 caracteres por polegada. Portanto, seu programa deve imprimir 6,5 polegadas (16,51 cm) de texto ou 65 caracteres por linha.

8.36 *(Imprimindo Datas em Vários Formatos)* As datas são impressas normalmente em vários formatos diferentes na correspondência comercial. Os dois formatos mais comuns são:

21/07/55 e 21 de julho de 1955

Escreva um programa que leia uma data no primeiro formato e imprima-a no segundo formato.

8.37 *(Proteção de Cheques)* Os computadores são empregados freqüentemente em sistemas de gravação de cheques como aplicativos de folhas de pagamento e contabilidade. Muitas histórias estranhas são contadas a respeito de cheques de pagamentos semanais serem impressos (por engano) com quantias superiores a 1 milhão de dólares. Quantias duvidosas são impressas por sistemas computadorizados de gravação de cheques devido a falhas humanas e/ou de máquina. Os projetistas de sistemas, obviamente, fazem todos os esforços para construir controles em seus sistemas e evitar a emissão de cheques errados.

Outro problema sério é a alteração intencional da quantia do cheque por alguém que pretenda descontá-lo fraudulentamente. Para evitar que a quantia seja alterada, a maioria dos sistemas computadorizados de gravação de cheques emprega uma técnica chamada *proteção de cheques.*

Os cheques destinados à impressão por computador contêm um número fixo de espaços nos quais o computador pode imprimir uma quantia. Suponha que um cheque para pagamento contenha oito espaços em branco nos quais o computador deve imprimir a quantia de um pagamento semanal. Se a quantia for grande, todos os oitos espaços serão preenchidos, por exemplo:

1,230.60 (quantia do cheque)
12345678 (números das posições)

Por outro lado, se a quantia for menor do que $1000, vários espaços serão deixados

originalmente em branco. Por exemplo,

99.87 12345678

contém três espaços em branco. Se for impresso um cheque com espaços em branco, fica mais fácil alguém alterar a quantia impressa. Para evitar que um cheque seja alterado, muitos sistemas de gravação de cheques inserem *asteriscos iniciais* para proteger a quantia, como se segue:

*****99.87 12345678**

Escreva um programa que receba um valor de uma quantia a ser impressa em um cheque e depois imprima a quantia no formato de proteção de cheques, com os asteriscos iniciais, se necessário. Admita que há nove espaços disponíveis para a impressão da quantia.

8.38 (E*screver a Palavra Equivalente a uma Quantia de um Cheque)* Continuando a análise do exemplo anterior, reiteramos a importância de desenvolver sistemas de gravação de cheques que evitem a alteração das quantias. Um método comum de segurança exige que a quantia seja escrita em números e em palavras. Mesmo que alguém seja capaz de modificar o valor numérico do cheque, é extremamente difícil modificar a quantia por extenso.

Muitos sistemas computadorizados de gravação de cheques não imprimem a quantia por extenso. Talvez o motivo principal dessa omissão seja o fato de que a maior parte das linguagens de alto nível usada em aplicativos comerciais não contém recursos adequados de manipulação de strings. Outra razão é que a lógica de escrever palavras equivalentes a quantias de cheques é um tanto complicada.

Escreva um programa em C que receba um valor numérico de um cheque e escreva a palavra equivalente. Por exemplo, o valor 112.43 deve ser escrito como

CENTO E DOZE e 43/100

8.39 (C*ódigo Morse)* Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para uso em sistemas telegráficos. O código Morse atribui uma série de pontos e traços a cada letra do alfabeto, a cada dígito e a alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-virgula). Em sistemas sonoros, o ponto representa um som breve e o traço representa um som longo.

Outras representações de pontos e traços são usadas com sistemas luminosos e baseados em bandeiras.

A separação entre palavras é indicada por um espaço ou, simplesmente, pela ausência de um ponto ou um traço. Em um sistema sonoro, um espaço é indicado por um período breve de tempo durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Fig. 8.39.

Escreva um programa que leia uma frase em português e traduza a para o código Morse. Escreva também um programa que leia uma frase em código Morse e converta-a em sua equivalente em português. Use um espaço em branco entre cada letra do código Morse e três espaços em branco entre palavras naquele código.

8.40 *(Programa de Conversão Métrica)* Escreva um programa que ajudará o usuário nas conversões métricas. Seu programa deve permitir que o usuário especifique os nomes das unidades como strings (i.e., centímetros, litros, gramas, etc. para o sistema métrico e polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

"Quantas libras ha em 2 metros?" "Quantos litros ha em 10 quartos?"

Seu programa deve reconhecer conversões inválidas. Por exemplo, a pergunta

"Quantos pes ha em 5 quilogramas?"

não faz sentido porque "pe" é unidade de comprimento enquanto "quilograma" é unidade de peso.

8.41 *(Cartas de Cobrança)* Muitas empresas gastam grande parte de tempo e dinheiro cobrando dívidas vencidas. *Cobrança **é*** o processo de fazer solicitações repetidas e insistentes a um devedor para tentar receber uma dívida

Caractere	Código	Caractere	Código
A	.–	T	–
B	–...	U	. . –
C	–.–.	V	. . . –
D	–. .	W	.– –
E	.	X	– . . –
F	. .–.	Y	–.– –
G	– – .	Z	– – . .
H	. . . .	Dígito	
I	. .	1	.– – – –
J	.– – –	2	. . – – –
L	. – . .	3	. . . – –
M	– –	4	. . . .–
N	– .	5	. . . . .
O	– – –	6	– . . . .
P	.– –.	7	– – . . .
Q	– –.–	8	– – – . .
R	.–.	9	– – – –.
S	. . .	0	– – – – –

Fig. 8.39 As letras do alfabeto expressas no código Morse Internacional.

Os computadores são usados freqüentemente para gerar cartas de cobrança automaticamente e com um grau crescente de severidade à medida que uma dívida permanecer. A teoria é que quanto mais velha se torna a dívida, mais difícil de cobrar ela se torna e, portanto, as cartas de cobrança devem ficar mais ameaçadoras.

Escreva um programa em C que possua os textos de cinco cartas de cobrança com severidade crescente. Seu programa deve aceitar como entrada:

1. O nome do devedor

2. O endereço do devedor
3. A conta do devedor
4. A quantia devida
5. O tempo da dívida (um mês, dois meses etc).

Use o tempo da dívida para selecionar um dos cinco textos de mensagens e então imprima a carta de cobrança inserindo as outras informações fornecidas quando apropriado.

Um projeto complexo de manipulação de strings

8.42 (*Um Gerador de Palavras Cruzadas)* A maioria das pessoas já fez palavras cruzadas uma vez ou outra, mas poucas já tentaram gerar uma. Gerar palavras cruzadas é um problema difícil. Ele é sugerido aqui como um projeto de manipulação de strings que exige grande esforço e sofisticação. Há muitas questões que o programador deve resolver para fazer com que até o mais simples programa gerador de palavras cruzadas funcione. Por exemplo, como é representada uma grade de palavras cruzadas no computador? Deve-se usar uma série de strings ou arrays bidimensionais? O programador precisa de uma fonte de palavras (i.e., um dicionário computadorizado) que possa ser referenciada diretamente pelo programa. Em que forma essas palavras devem ser armazenadas para facilitar a manipulação complexa exigida pelo programa? O leitor verdadeiramente ambicioso desejará gerar a parte de "pistas" das palavras cruzadas na qual as instruções resumidas de cada palavra horizontal e vertical são impressas para quem está procurando resolver o quebra-cabeças. Apenas imprimir uma versão de palavras cruzadas em branco não é um problema simples.

9

Formatação de Entrada/Saída

Objetivos

- Entender os fluxos de entrada e saída.
- Ser capaz de usar todos os recursos de formatação de impressão.
- Ser capaz de usar todos os recursos de formatação de entrada.

Todas as notícias que sejam adequadas para impressão.
Adolph S. Ochs

Que perseguição louca? Que luta para escapar?
John Keats

Não remova o marco no limite dos campos.
Amenemope

O fim deve justificar os meios.
Matthew Prior

Sumário

9.1 Introdução

Uma parte importante da solução de qualquer problema é a apresentação dos resultados. Neste capítulo analisaremos em profundidade os recursos de formatação de **printf** e **scanf.** Estas funções recebem dados do *fluxo de entrada padrão* e enviam dados para o *fluxo de saída padrão,* respectivamente. Quatro outras funções que usam o dispositivo-padrão de entrada e o dispositivo-padrão de saída — **gets, puts, getchar** e **putchar** — foram vistas no Capítulo 8. Inclua o arquivo de cabeçalho **<stdio.h> em programas** que chamam essas funções.

Muitos recursos de **printf** e **scanf** foram analisados anteriormente no texto. Este capítulo resume aqueles recursos e apresenta muitos outros. O Capítulo 11 analisa várias outras funções incluídas na biblioteca-padrão de entrada/saída **(stdio).**

9.2 Fluxos (Streams)

Todas as entradas e saídas são realizadas com *fluxos (streams)* — seqüências de caracteres organizados em linhas. Cada linha consiste em zero ou mais caracteres e termina com um caractere de nova linha. O padrão preconiza que as implementações em ANSI C devem suportar linhas de pelo menos 254 caracteres incluindo um caractere de nova linha para terminação. Quando a execução de um programa é iniciada, três fluxos são conectados automaticamente ao programa. Normalmente, o fluxo de entrada padrão é conectado ao teclado e o fluxo de entrada padrão está conectado à tela. Freqüentemente, os sistemas operacionais permitem que esses fluxos sejam redirecionados para outros dispositivos. Um terceiro fluxo, *erro-padrão,* é conectado à tela. As mensagens de erro são enviadas para o fluxo de erro-padrão. Os fluxos são analisados com mais detalhes no Capitulo 11, "Processamento de Arquivos".

9.3 Formatação da Saída com Printf

A formatação precisa da saída é realizada com **printf.** Toda chamada a **printf** contém uma *string de controle de formato* que descreve o formato da saída. A string de controle de formato consiste nos especificadores *de conversão, sinalizadores (flags), larguras de campo* e *caracteres literais.* A função printf pode realizar as seguintes ações de formatação e cada uma delas é analisada neste capítulo.

1. *Arredondamento* de valores de ponto flutuante para um número indicado de casas decimais.

2. *Alinhamento* de uma coluna de números por seus pontos decimais aparecendo sobrepostos.

3. *Justificação à direita* e *à esquerda* das saídas.

4. *Inserção de caracteres literais* em locais precisos de uma linha na saída.

5. Representação de números de ponto flutuante no formato exponencial.

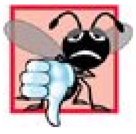
6. Representação de inteiros sem sinal (unsigned) no formato octal e hexadecimal. Veja o Apendice D, "Sistemas de Numeração" para obter mais informações sobre valores octais e hexadecimais.

7. Apresentação de todos os tipos de dados com larguras de campos de tamanho fixo e com precisão.

A função **printf** tem a forma:

printf (*string de controle de formato, outros argumentos);*

A *string de controle de formato* descreve o formato da saída, e *outros argumentos* (esses são opcionais correspondem a cada especificação de conversão na *string de controle de formato.* Cada especificação de conversão começa com um sinal de percentagem e termina com um especificador de conversão. Pode haver muitas especificações de conversão em uma string de controle de formato.



Erro comun de programação 9.1

Esquecer de colocar a string de controle de formato entre aspas duplas.



Boa prática de programação 9.1

Edite concisamente as saídas para criar apresentações. Isso torna as saídas do programa mais legíveis e reduz os erros dos usuários.

9.4 Imprimindo Inteiros

Um inteiro é um número, como 776 ou - 52, que não possui ponto decimal. Os valores inteiros são exibidos em um dentre vários formatos. A Fig. 9.1 descreve cada um dos especificadores de conversão de inteiros.

O programa da Fig. 9.2 imprime um inteiro usando cada um de seus especificadores de conversão. Observe que apenas os sinais negativos são impressos; os sinais positivos são omitidos. Mais adiante neste capítulo veremos como fazer com que os sinais positivos sejam impressos. Observe também que o valor **-455 é l**ido por %u e convertido para o valor sem sinal **65081** em um computador com inteiros de 2 bytes.

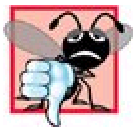
Especificador de conversão	**Descrição**
d	Exibe um inteiro decimal.
i	Exibe um inteiro decimal com sinal (Nota: Os especificadores **i** e **d** são diferentes ao ser usados com **scanf**.)
o	Exibe um inteiro octal sem sinal(unsigned)
x ou X	Exibe um inteiro hexadecimal sem sinal. **X** faz com que os dígitos **0 – 9** e as letras **a – f.**
h ou **l** (letra l)	Colocar antes de qualquer especificador de conversão de inteiros para indicar que um inteiro **short** ou **long** deve ser exibido, respectivamente.

Fig. 9.1 Especificadores de conversão de inteiros.

```
/* Usando os especificadores de conversão de inteiros */
#include <stdio.h>
main() {
printf("%d\n", 455);
printf("%i\n",  455);     /* i faz o mesmo que d em printf */
printf("%d\n", +455);
printf("%d\n", -455);
printf("%hd\n", 32CkO0);
printf("%ld\n", 2000000000);
printf("%o\n", 455);
printf("%d\n" , 455);
printf("%u\n", -455);
printf("%x\n", 455);
printf("%X\n", 455);
return 0;
}
```

```
455
455
455
-455
32000
2000000000
707
455
65081
lc7
1C7
```

Fig. 9.2 Usando os especificadores de conversão de inteiros.



Erro comun de programação 9.2

Imprimir um valor negativo com um especificador de conversão que aguarda um valor sem sinal (unsigned).

9.5 Imprimindo Números de Ponto Flutuante

Um valor de ponto flutuante contém um ponto decimal como 33.5 ou 657.983. Os valores de ponto flutuante são exibidos em um dentre vários formatos. A Fig. 9.3 descreve os especificadores de conversã**o** de ponto flutuante.

Os especificadores de conversão **e** e **E** exibem valores de ponto flutuante na *notação exponencial.* A notação exponencial é o equivalente computacional à *notação científica* usada na matemática. Por exemplo**,** o valor 150.4582 é representado em notação científica como

1.504582 X 10^2

e é representado em notação exponencial como

1.504582E+02

pelo computador. Essa notação indica que 1.504582é multiplicado por 10 elevado à segunda potência **(E**+02). O **E** indica "expoente".

Especificador de conversão	Descrição
e ou **E**	Exibe um valor de ponto flutuante em notação exponencial.
F	Exibe valores de ponto flutuante.
g ou **G**	Exibe um valor de ponto flutuante tanto na forma de ponto flutuante **f** quanto na forma exponencial **e** (ou **E).**
L	Colocar antes de qualquer especificador de ponto flutuante para indicar que um valor de ponto flutuante l**ong double** será exibido.

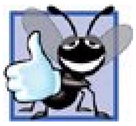
Fig. 9.3 Especificadores de conversão de ponto flutuante.

Os valores impressos com os especificadores de conversão **e, E** e **f** são impressos com 6 dígitos de precisão à direita do ponto decimal por default; outras precisões podem ser especificadas explícitamente. O especificador de conversão **f** sempre imprime pelo menos um dígito à esquerda do ponto decimal. Os especificadores de conversão **e** e **E** imprimem a letra minúscula **e** e a letra maiuscula E antes do expoente, respectivamente, e sempre imprimem exatamente um dígito à esquerda do ponto decimal.

O especificador de conversão **g (G)** imprime tanto o formato **e (E)** quanto o **f** sem zeros finais (i.e., **1.234000** é impresso como **1.234).** Os valores são impressos com **e (E)** se depois de converter o valor para a notação exponencial. o expoente do valor for menor do que **-4** ou for maior ou igual a precisão especificada (6 dígitos significativos por default, para **g** ou **G).** Caso contrário, o especificador de conversão **f** é usado para imprimir o valor. Os zeros finais não são impressos na parte fracionária de saída de um valor com **g** ou **G.** Pelo menos um dígito decimal é exigido para o ponto decimal ser

enviado para a saída. Os valores **0.0000875, 8750000.0 0, 8.75, 87.50** e **875** são impressos como **8.75e-05, 8.75e+06, 8.75, 87.5** e **875** com a especificação de conversão %g. O valor **0.0000875** usa a notação **e** porque, quando ele é convertido para a notação exponencial, seu expoente é menor que **-4. O** valor **8750000.0** usa a notação **e** porque seu expoente é igual à precisão default.

A precisão para os especificadores de conversão **g** e **G** indica o número máximo de dígitos significativos impressos incluindo o dígito à esquerda do ponto decimal. O valor **1234567.0** é impresso como **1.234567e+06** usando a especificação de conversão %***g*** (lembre-se de que todos os especificadores de ponto flutuante possuem precisão default de 6). Observe que há 6 dígitos significativos no resultado. A diferença entre **g** e **G** é idêntica à diferença entre **e** e **E** quando o valor é impresso na notação exponencial — a letra minúscula **g** faz com que uma letra minúscula **e seja** enviada para o dispositivo de saída e a letra maiúscula **G** faz com que uma letra maiúscula **E seja** enviada para o dispositivo de saída.



Boa prática de programação 9.2

Ao enviar dados para o dispositivo de saída, certifique-se de que o usuário está ciente das situações nas quais os dados podem estar imprecisos devido à formatação (e.g., erros de arredondamento de precisões específicas).

O programa da Fig. 9.4 demonstra cada uma das três especificações de conversão de ponto flutuante. Observe que as especificações de conversão %E e %**g** fazem com que o valor seja arredondado na saída.

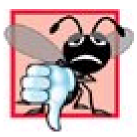
9.6 Imprimindo Strings e Caracteres

Os especificadores de conversão **c** e **s** são usados para imprimir caracteres isolados e strings, respectivamente. O especificador de conversão **c** exige um argumento **char. O** especificador de conversão **s** exige como argumento um ponteiro para **char. O** especificador de conversão **s** faz com que os caracteres sejam impressos até que seja encontrado um caractere NULL(' \ 0 '). O programa mostrado na Fí**g.** 9.5 apresenta caracteres e strings com os especificadores de conversão **c** e **s.**

```
/* Imprimindo números de ponto flutuante
com especificadores de conversão de ponto flutuante */
#include <stdio.h>
main() {
printf("%e\n", 1234567.89);
printf("%e\n", +1234567.89);
printf("%e\n", -1234567.89);
printf("%E\n", 1234567.89);
printf("%f\n", 1234567.89);
printf("%g\n", 1234567.89);
printf("%G\n", 1234567.89);
return 0;
}
```

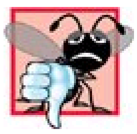
```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Fig. 9.4 Usando os especificadores de conversão de ponto flutuante



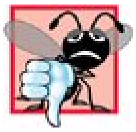
Erro comun de programação 9.3

*Usar %c para imprimir o primeiro caractere de uma string. A especificação de conversão %***c** *exige um argumento char. Uma string é um ponteiro para char, i.e., um char *.*



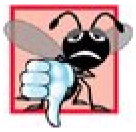
Erro comun de programação 9.4

Usar %s para imprimir um argumento char. A especificação de conversão %s exige um argumento do tipo ponteiro para char. Em alguns sistemas, isso causa um erro fatal em tempo de execução chamado violação de acesso.



Erro comun de programação 9.5

Usar aspas simples em torno de strings de caracteres é um erro de sintaxe. As strings de caracteres devem ser colocadas entre aspas duplas.



Erro comun de programação 9.6

Usar aspas duplas em torno de uma constante de caractere. Isso cria realmente uma string constituída de dois caracteres, sendo o segundo deles o caractere NULL de terminação. Uma constante de caracteres é um único caractere entre aspas simples.

```
/* Imprimindo strings e caracteres */
#include <stdio.h>

main() {

char character = 'A';
char string[]  =  "Isto e uma string";
char *stringPtr = "Isto também e uma string";

printf("%c\n", character);
printf("%s\n",   "Isto e uma string");
printf("%s\n", string);
printf("%s\n",  stringPtr);

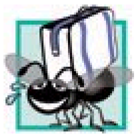
return 0;
}
```

```
Isto e uma string Isto e uma string Isto também e uma string
```

Fig. 9.5 Usando os especificadores de conversão de caracteres e strings.

9.7 Outros Especificadores de Conversão

Os três especificadores de conversão restantes são **p, n** e % (Fig. 9.6).



Dicas de portabilidade 9.1

O especificador de conversão p exibe um endereço de ponteiro de uma forma dependente da implementação (em muitos sistemas a notação hexadecimal é utilizada em vez da notação decimal).

O especificador de conversão **n** armazena o número de caracteres já enviados ao dispositivo de saída na instrução **printf** — o argumento correspondente é um ponteiro para uma variável inteira na qual o valor está armazenado. Nada é impresso por uma especificação de conversão %**n. O** especificador de conversão % faz com que um sinal de porcentagem seja enviado ao dispositivo de saída.

No programa da Fig. 9.7 %**p** imprime o valor **ptr** e o endereço de **x;** esses valores são idênticos porque foi atribuído a **ptr** o valor de **x.** A seguir, %n armazena na variável **y** o número de caracteres enviado ao dispositivo de saída pela terceira instrução **printf** e o valor de **y** é impresso. A última instrução **printf** usa %% para imprimir o caractere % em uma string de caracteres. Observe que chamada a **printf** retoma um valor — seja o número de caracteres enviados ao dispositivo de saída, seja um valor negativo, se ocorrer um erro.

Especificador de conversão	**Descrição**
P	Exibe um valor de ponteiro de acordo com implementação.
n	Armazena o número de caracteres já enviados para o dispositivo de saída na instrução **printf** **a**tual. Um ponteiro para um inteiro é fornecido como o argumento correspondente. Nada é exibido.
%	Exibe o caractere de porcentagem.

Fig. 9.6 Outros especificadores de conversão.

```c
/* Usando os especificadores de conversão p, n e % */
#include <stdio.h>

main() {

int *ptr;
int x = 12345, y;
ptr = &x; 3

printf("O valor de ptr e %p\n", ptr);
printf("O endereço de x e %p\n\n", &x);
printf("O total de caracteres impresso nesta linha e:%n",  &y);
printf(" %d\n\n", y) ;

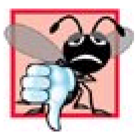
y = printf("Esta linha tem 28 caracteres\n");

printf("%d caracteres foram impressos\n\n", y);
printf("Imprimindo um %% em um formato de string de controle\n");

return 0;
}
```

```
O valor de ptr e 001F2BB4
O endereço de x e 001F2BB4
O total de caracteres impresso nesta linha e: 45
Esta linha tem 28 caracteres 28 caracteres foram impressos
Imprimindo um % em um formato de string de controle
```

Fig. 9.7 Usando os especificadores de conversão p, n e %.

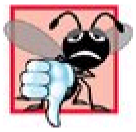


Erro comun de programação 9.7

Tentar imprimir um caractere de porcentagem usando % em vez de %% na string de controle de formato. Quando % aparece em uma string de controle de formato, um especificador de conversão deve vir em seguida.

9.8 Imprimindo com Larguras de Campos e Precisões

O tamanho exato de um campo no qual os dados são impressos é especificado por um *tamanho (largura) de campo.* Se o tamanho do campo for maior do que os dados a serem impressos, normalmente esses serão justificados à direita naquele campo. Um inteiro representando o tamanho do campo é inserido entre o sinal de porcentagem (%) e o especificador na especificação de conversão. O programa da Fig. 9.8 imprime dois grupos de cinco números, justificando à direita os números que possuírem menos dígitos do que o tamanho do campo. Observe que o tamanho do campo é aumentado automaticamente para imprimir valores maiores do que o campo e que o sinal de subtração de um valor negativo usa uma posição de caractere no tamanho do campo. Os tamanhos de campos podem ser usados com todos especificadores de conversão.



Erro comun de programação

Não fornecer um tamanho de campo suficientemente grande para manipular o valor a ser impresso. Isso pode compensar a impressão de outros dados e pode produzir saídas confusas. Conheça seus dados!

```
/* Imprimindo inteiros alinhados pela direita */
#include <stdio.h>

main() {

printf("%4d\n", 1);
printf("%4d\n", 12);
printf("%4d\n", 123);
printf("%4d\n", 1234);
printf("%4d\n\n", 12345);
printf("%4d\n", -1);
printf("%4d\n", -12);
printf("%4d\n", -123);
printf("%4d\n", -1234);
printf("%4d\n", -12345);

return 0;

}
```

```
1 12 123 1234 12345
-1 -12 -123 -1234 -12345
```

Fig. 9.8 Alinhando inteiros pela direita em um campo.

A função **printf** também fornece a capacidade de especificar a *precisão* com a qual os dados são impressos. Precisão tem significados diferentes para diferentes tipos de dados. Ao ser usada com especificadores inteiros de conversão, precisão indica o número mínimo de dígitos a serem impressos. Se o valor impresso possuir menos

dígitos do que o especificado na precisão, são colocados zeros à frente do valor impresso até que o número total de dígitos seja equivalente à precisão. A precisão default para inteiros é 1. Ao ser usada com os especificadores de ponto flutuante **e, E** e f. a precisão é o número de dígitos que aparece depois do ponto decimal. Ao ser usada com os especificadores **g** e **G,** a precisão é o número máximo de dígitos a serem impressos. Ao ser usada com o especificador de conversão **s,** a precissão é o número máximo de caracteres da string a serem escritos. Para usar precisão, coloque um ponto decimal (.) seguido de um inteiro representando a precisão entre o sinal de porcentagem e o especificador de conversão. O programa da Fig. 9.9 demonstra o uso da precisão nas strings de controle de formato. Observe que quando um valor de ponto flutuante é impresso com uma precisão menor do que o número original de casas decimais no valor, este é arredondado.

O tamanho do campo e a precisão podem ser combinados colocando o tamanho do campo seguido de um ponto decimal e colocando a seguir a precisão entre o sinal de porcentagem e o especificador de conversão, como na instrução

printf("%9.3f", 123.456789);

que imprime **123.457** com três dígitos à direita do ponto decimal e justificado à direita em um campo de 9 dígitos.

```
/* Usando precisão ao imprimir inteiros,
números de ponto flutuante e strings */
#include <stdio.h>

main() {

int i = 873;
float f = 123.94536;
char s[]  =  "Happy Rirthday";

printf("Usando precisão para inteiros\n");
printf("\t%.4d\n\t%.9d\n\n",  i, i);
printf("Usando precisão para números de ponto flutuante\n");
printf("\t%.3f\n\t%.3e\n\t%.3g\n\n",  f,  f,  f) ;
printf("Usando precisão para strings\n");
printf("\t%.lls\n", s);

return 0;
}
```

```
Usando precisão para inteiros 0873
000000873
Usando precisão para números de ponto flutuante 123.945 1.239e+02 124
Usando precisão para strings Happy Birth
```

Fig. 9.9 Usando precisões para exibir informações de vários tipos

É possível especificar o tamanho do campo e a precisão usando expressões inteiras na lista de argumentos após a string de controle de formato. Para usar esse recurso, insira um * (asterisco) no lugar do tamanho do campo ou da precisão (ou de ambos). O argumento correspondente da lista é calculado e usado no lugar do asterisco. O valor do argumento pode ser negativo para o tamanho do campo mas deve ser positivo para a precisão. Um valor negativo para o tamanho de campo faz com que a saída seja justificada à esquerda no campo como descreve a próxima seção. A instrução

printf("%*.*f", 7, 2, 98.736);

usa 7 para tamanho de campo, **2** para precisão e envia o valor **98.74,** justificado à direita, para o dispositivo de saída.

9.9 Usando Sinalizadores (Flags) na String de Controle de Formato de Printf

A função **printf** também fornece *sinalizadores (flags)* para suplementar seus recursos de formatação. Há cinco sinalizadores disponíveis para uso do usuário em strings de controle de formato (Fig. 9.10).

Sinalizador (Flag)	Descrição
- (sinal de subtração)	Alinha a saída pela esquerda no campo especificado.
+(sinal de adição	Exibe um sinal de adição antes de valores positivos e um sinal de subtração antes e valores negativos
Espaço	Imprime um espaço antes de um valor positivo não impresso com o sinalizador +.
#	Coloca um zero antes do valor de saída quando usado com o especificador de conversão octal o. Coloca 0x ou 0X antes do valor de saída quando usado com os especificadores de conversão hexadecimais x ou X Coloca obrigatoriamente o ponto decimal em um numero de ponto flutuante impresso com e, E, f, g ou G que não contém parte fracionária.(Normalmente o ponto decimal só é impresso se vier seguido de um dígito.). Para os especificadores g ou G, os zeros finais não são eliminados.
0 (zero)	Preenche um campo com zeros iniciais.

Fig. 9.10 Sinalizadores (flags) de string de controle de formato.

Para usar um sinalizador em uma string de controle de formato, coloque-o imediatamente à direita do sinal de porcentagem. Vários sinalizadores podem ser combinados em uma especificação de conversão.

O programa da Fig. 9.11 demonstra a justificação à direita e à esquerda de uma string, de um inteiro de um caractere e de um número de ponto flutuante.

```
/* Alinhando valores pela esquerda e pela direita */
#include <stdio.h>
main(){

printf("%10s%10d%10c%10f\n\n",   "hello",  7,   'a', 1.23);
printf("%-10s%-10d%-10c%-10f\n",   "hello",  7,   'a', 1.23);

return 0;
}
```

```
hello 7 a 1.230000
hello 7 a 1.230000
```

Fig. 9.11 Alinhando (justificando) strings pela esquerda em um campo.

O programa da Fig. 9.12 imprime um número positivo e um número negativo, cada um deles com e sem o sinalizador +. Observe que o sinal de menos é exibido em ambos os casos, mas o sinal de mais é apresentado quando o sinalizador + é utilizado.

```
/* Imprimindo números com e sem o sinalizador + */
#include <stdio.h>

main() {

printf("%d\n%d\n",   786, -786);
printf("%+d\n%+d\n",  786, -786);

return 0;
}
```

```
786 -786 + 786 -786
```

Fig. 9.12 Imprimindo números positivos e negativos com e sem o sinalizador +.

O programa da Fig. 9.13 coloca um espaço à frente de um número positivo utilizando o sinalizador de espaço. Isso é útil para alinhar números positivos e negativos com o mesmo numero de digitos.

```
/* Imprimindo um espaço antes de valores com sinal
nao antecedidos por + ou - */
#include <stdio.h>

main() {

printf("% d\n% d\n",   547, -547);
return 0;
}
```

```
547 -547
```

Fig. 9.13 Usando o sinalizador de espaço.

O programa da Fig. 9.14 usa o sinalizador # para colocar um 0 à frente de um valor octal, **0x** e **0X** frente de valores hexadecimais e para impor o ponto decimal em um valor impresso com **g.**

```
/* Usando o sinalizador # com os especificadores de conversão
o, x, X e qualquer outro especificador de ponto flutuante */
#include <stdio.h>

main() {

int c = 1427;
float p = 1427.0;

printf("%#o\n", c);
printf("%#x\n", c);
printf("%#X\n", c);
printf("\n%g\n", p);
printf("%#g\n", p);

return 0;
}
```

```
02623 0x593 0X593
1427 1427.00
```

Fig. 9.14 Usando o sinalizador #.

O programa da Fig. 9.15 combina o sinalizador + com o sinalizador 0 para imprimir 452 em um campo de 9 espaços com um sinal + e com zeros iniciais e a seguir imprime 452 novamente usando apenas o sinalizador 0 e um campo de 9 dígitos.

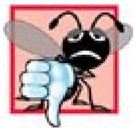
```
/* Imprimir com o sinalizador 0 (zero) insere zeros iniciais */
#include <stdio.h>
maín(){
printf("%+09d", 452);
printf("%09d", 452);
return 0;
}
```

```
+00000452 000000452
```

Fig. 9.15 Usando o sinalizador o (zero).

9.10 Imprimindo Seqüências Literais e de Escape

A maior parte dos caracteres literais a serem impressos em uma instrução printf pode simplesmente ser incluída na string de controle de formato. Entretanto, há vários caracteres "problemáticos" com as aspas duplas (") que delimita a própria string de controle de formato. Vários caracteres de controle, como nova linha e tabulação, devem ser representados por *seqüências de escape.* Uma seqüência de escape é representada por uma barra invertida (\, chamada *backslash.* em inglês) seguida de um *caractere de escape* específico. A tabela da Fig. 9.16 lista todas as seqüências de escape e as ações que elas causam.



Erro comun de programação 9.9

Tentar imprimir aspas simples, aspas duplas, um sinal de interrogação ou uma barra invertida como dado literal em uma instrução printf sem preceder aquele caractere de uma barra invertida para forma seqüência de escape apropriada.

9.11 Formatação da Entrada com Scanf

A formatação precisados dados de entrada é realizada com **scanf.** Toda instrução **scanf** contém uma string de controle de formato que descreve o formato dos dados a serem fornecidos. A string de controle de formato consiste em especificações de conversão e caracteres literais. A função **scanf** oferece os seguintes recursos de formatação dos dados de entrada:

1. Aceita todos os tipos de dados.
2. Aceita caracteres específicos de um fluxo de entrada.
3. Ignora caracteres específicos de um fluxo de entrada.

Seqüência de escape	**Descrição**
\'	Imprime um caractere de aspas simples (').
\"	Imprime um caractere dc aspas duplas (").
\?	Imprime o sinal de interrogação (?).
\\	Imprime o caractere de barra invertida (backslash, \).
\a	Emite um sinal de alerta sonoro (sino) ou visual.
\b	Move o cursor uma posição para trás na linha atual.
\f	Move o cursor para o início da próxima página lógica.
\n	Move o cursor para o início da próxima linha.
\r	Move o cursor para o inicio da linha atual.
\t	Move o cursor para a próxima posição de tabulação horizontal.
\v	Move o cursor para a próxima posição de tabulação vertical.

Fig. 9.16 Seqüências de escape.

Especificador de conversão	**Descrição**
Inteiros	
d	Lê um inteiro decimal com sinal opcional. O argumento correspondente é um ponteiro para um inteiro.
i	Lê um inteiro decimal, octal ou hexadecimal com sinal opcional. 0 argumento correspondente é um ponteiro para um inteiro.
o	Lê um inteiro octal. O argumento correspondente é um ponteiro para um inteiro sem sinal.
u	Lê um inteiro decimal sem sinal. O argumento correspondente é um ponteiro para um inteiro sem sinal.
x ou X	Lê um inteiro hexadecimal. O argumento correspondente e um ponteiro para um inteiro sem sinal.
h ou 1	Colocar antes de qualquer especificador de conversão de inteiros para indicar que um inteiro **short** ou l**ong** deve ser fornecido como dado de entrada.
Números de ponto flutuante	
e, E, f, g ou **G**	Lê um valor de ponto flutuante. 0 argumento correspondente é um ponteiro para uma variável de

	ponto flutuante.
l ou L	Colocar antes de qualquer especificador de conversão de ponto flutuante para indicar que um valor **double** ou **long double** deve ser fornecido como dado de entrada.
Caracteres e strings	
c	Lê um caractere. O argumento correspondente é um ponteiro para **char,** não é adicionado nenhum null ('\0').
s	Lê uma string. O argumento correspondente é um ponteiro para um array do tipo **char** que é suficientemente grande para conter a string e um caractere null ('\ 0') de terminação.
Conjunto de varredura	
[caracteres de varredura]	Percorre uma string à procura de um conjunto de caracteres que estão armazenados em um array.
Vários	
p	Lê um endereço de ponteiro do mesmo tipo que o produzido quando um endereço é impresso com %*p* em uma instrução **printf**
n	Armazena o número de caracteres fornecidos como dados de entrada até esse ponto na instrução **scanf** atual. O argumento correspondente é um ponteiro para um inteiro.
%	Ignora um sinal de percentagem (%s) nos dados de entrada.

Fig. 9.17 Especificadroes de conversão para scanf.

A função **scanf** tem a seguinte forma:

scanf *(string de controle de formato, outros argumentos)* .

A string de controle de formato descreve os formatos dos dados de entrada, e *outros argumentos* são ponteiros a variáveis nas quais os dados de entrada são armazenados.



Boa prática de programação 9.3

Durante a entrada de dados, peça ao usuário um item de dados ou alguns itens de dados por vez; Evite pedir ao usuário para entrar com muitos itens de dados em resposta a um único pedido.

A Fig. 9.17 mostra um resumo dos especificadores de conversão usados para entrada de todos os tipos de dados. O restante desta seção fornece programas que demonstram a leitura de dados com os vários especificadores de conversão de **scanf.**

```
/* Lendo inteiros */
#include <stdio.h>
main(){

int a, b, c,  d,  e,  f, g;

printf("Digite sete inteiros: ");
scanf ("%d%i%i%i%o%u%x.",  &a,  &b, &c,  &d,  &e,  &f,  &g);
printf("Os dados de entrada exibidos como inteiros decimais sao: r.
printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);

return 0;
}
```

```
Digite sete inteiros:  -70 -70 070 0x70 70 70 70
Os dados de entrada exibidos como inteiros decimais sao:
-70 -70 56 112 56 70 112
```

Fig. 9.18 Lendo dados de entrada com os especificadores de conversão de inteiros**.**

O programa da Fig. 9.18 lê inteiros com os vários especificadores de conversão de inteiros e exibe os inteiros como números decimais. Observe que %i é capaz de receber inteiros decimais, octais e hexadecimais como dados de entrada.

Ao entrar com números de ponto flutuante, qualquer um dos especificadores de conversão de ponto flutuante, **e, E, f**, g ou **G,** pode ser utilizado. O programa da Fig. 9.19 demonstra a leitura de três números de ponto flutuante, cada um deles com um dos três tipos de especificador de conversão de ponto flutuante e exibe todos os números com o especificador de conversão **f**. Observe que a saída do programa confirma que os valores de ponto flutuante são imprecisos — isso é destacado pelo segundo valor impresso.

Caracteres c strings são fornecidos como dados de entrada por meio dos especificadores de conversão **c** e **s,** respectivamente. O programa da Fig. 9.20 pede ao usuário que digite uma string. O programa recebe o primeiro caractere da string com *%c* e armazena-o na variável de caracteres **x.** A seguir, o programa recebe o restante da string com *%s* e armazena-o no array de caracteres **y.**

Uma seqüência de caracteres pode ser fornecida usando um *conjunto de varredura (scan set). Um* conjunto de varredura é um conjunto de caracteres colocado entre colchetes [ ] e precedidos por um sinal de porcentagem na string de controle de formato. Um conjunto de varredura percorre os caracteres no fluxo de entrada a procura apenas pelos caracteres iguais àqueles contidos no conjunto de varredura. Cada vez que um caractere é encontrado, ele é armazenado no argumento correspondente do conjunto de varredura — um ponteiro para um array de caracteres. O conjunto de varredura cessa a leitura de caracteres quando for encontrado um caractere que não estiver contido naquele conjunto. Se o primeiro caractere do fluxo de entrada não for igual a nenhum caractere do conjunto de varredura, apenas o caractere nulo **(NULL)** será armazenado no array. O programa da Fig. 9.21 usa o conjunto de varredura [aeiou], para procurar por vogais no fluxo de entrada. Observe que as sete primeiras letras da entrada são lidas. A oitava letra **(h)** não está no conjunto e portanto a varredura é concluída.

O conjunto de varredura também pode ser usado para procurar por caracteres que não estejam contidos naquele conjunto por meio de um *conjunto de varredura invertido.* Para criar um conjunto de varredura invertido, coloque um *circunflexo* (^) entre os colchetes, antes dos caracteres do conjunto. Isso faz com que os caracteres que não apareçam no conjunto de varredura sejam armazenados. Quando for encontrado um caractere contido no conjunto de varredura invertido, a entrada de dados é encerrada. O programa da Fig. 9.22 usa o conjunto de varredura invertido [A**aeiou]** para procurar mais adequadamente por consoantes — para procurar por "não-vogais".

Um tamanho de campo pode ser usado em uma especificação de conversão **scanf** para ler um número específico de caracteres de um fluxo de entrada. O programa da Fig. 9.23 recebe uma série de dígitos consecutivos como um inteiro de dois dígitos e um inteiro consistindo nos dígitos remanescentes do fluxo de

```
/* Lendo números de ponto flutuante */
#include <stdio.h>

main() {

float a, b, c;

printf("Digite tres números de ponto flutuante:  \n");
scanf("%e%f%g",  &a,&b, &c);
printf("Eis os números fornecidos na notação comum\n");
printf("de ponto flutuante:\n");
printf("%f %f %f\n",  a, b,  c);

return 0;
}
```

```
Digite tres números de ponto flutuante:
1.27987 1.27987e+03 3.38476e-06
Eis os números fornecidos na notação comum
de ponto flutuante:
1.279870
1279.869995
0.000003
```

Fig. 9.19 Lendo dados de entrada com os especificadores de ponto flutuante.

```
/* Lendo caracteres e strings */
#include <stdio.h>

main() {

char x, y[9];

printf("Digite uma string:   ");
scanf("%c%s",  &x, y);
```

```
printf("A entrada foi:\n");
printf("o caractere \"%c\"   ",  x);
printf("e a string\"%s\"\n", y);

return 0;
}
```

```
Digite uma string: Sunday
A entrada foi:
o caractere "S" e a string "unday"
```

Fig. 9.20 Fornecendo caracteres e strings como dados de entrada.

```
/* Usando um conjunto de varredura */
#include <stdio.h>

main ()

char z [9];

printf("Digite String: ");
scanf("%[aeiou]", z);
printf("A entrada foi \"%s\"\n", z);

return 0;
}
```

```
Digite String: ooeeóoaehahe
A entrada foi "ooeeooa"
```

Fig. 9.21 Usando um conjunto de varredura.

```
/* Usando um conjunto de varredura invertido */
#include <stdio.h>

main() {

char z[9];

printf("Digite uma string: ");
scanf("%[Aaeiou]", z);
printf("A entrada foi \"%s\"\n",  z);

return 0;
}
```

```
Digite uma string: String
A entrada foi "Str"
```

Fig. 9.22 Usando um conjunto de varredura invertido.

```
/* entrando dados com um tamanho de campo */
#include <stdio.h>

main(){

int x, y;

printf("Digite um inteiro de seis digitos: ");
scanf("%2d%d", &x,  &y) ;
printf("Os inteiros fornecidos foram %d e %d\n", x, y);

return 0;
}
```

```
Digite um inteiro de seis digitos: 123456
Os inteiros fornecidos foram 12 e 3456
```

Fig. 9.23 Entrando dados com um tamanho de campo.

Freqüentemente se faz necessário ignorar determinados caracteres do fluxo de entrada. Por exemplo, uma data poderia ser fornecida como

7-9-91

Cada número da data precisa ser armazenado, mas os travessões que separam os números podem ser descartados. Para eliminar os caracteres desnecessários, inclua-os na string de controle de formato de **scanf** (caracteres de espaço em branco — como espaço, nova linha e tabulação — fazem com que sejam ignorados todos os espaços em branco iniciais). Por exemplo, para ignorar os travessões na entrada da data, use a instrução

scanf(*"%d-%d-%d"***, &mes, &dia, &ano);**

Embora esse **scanf** elimine os travessões da entrada anterior, é possível que a data seja fornecida como

7/9/91

Nesse caso, o **scanf** precedente não eliminaria os caracteres desnecessários. Por esse motivo, **scanf** oferece o *caractere de supressão de atribuição* *. O caractere de supressão de atribuição permite que **scanf** leia qualquer tipo de dado de entrada e ignore-o sem atribuí-lo a uma variável. O programa da Fig. 9.24 usa o caractere de supressão de atribuição na especificação de conversão *%c* para indicar que um caractere

que apareça no fluxo de entrada deve ser lido e ignorado. Apenas o mês, dia e ano são armazenados. Os valores das variáveis são impressos para demonstrar que foram realmente recebidos corretamente. Observe que nenhuma variável da lista de argumentos corresponde às especificações de conversão que usam o caractere de supressão de atribuição porque não há atribuição realizada por aquelas especificações de conversão.

```
/* Lendo e ignorando caracteres do fluxo de entrada */
#include <stdio.h>

main() {

int mesl,  dial,  anol, mes2, dia2, ano2;

printf("Digite uma data na forma mm-dd-aa: ");
scanf("%d%*c%d%c%d",  &mesl,  &dial, &anol);
printf("mes = %d dia = %d ano = %d\n\n",mesl,  dial,  anol);
printf("Digite uma data na forma mm/dd/aa:   ");
scanf("%d%*c%d%*c%d",  &mes2,  &dia2, &ano2);
printf("mes = %d dia = %d ano = %d\n",mes2,  dia2,  ano2);

return 0;
}
```

```
Digite uma data na forma mm-dd-aa:
11-18-71 mes = 11 dia = 18 ano = 71

Digite uma data na forma mm/dd/aa:
11/18/71 mes = 11 dia = 18 ano = 71
```

Fig. 9.24 Lendo e ignorando caracteres de um fiuxo de entrada.

Resumo

• Toda entrada e saída é realizada por meio de fluxos (streams) — seqüências de caracteres organizadas em linhas. Cada linha consiste em zero ou mais caracteres e termina com um caractere de nova linha.
• Normalmente, o fluxo de entrada padrão está conectado ao teclado e o fluxo de saída padrão está conectado à tela do computador.
• Os sistemas operacionais permitem freqüentemente que os fluxos dos dispositivos-padrão de entrada e saída sejam redirecionados para outros dispositivos.
• A string de controle de formato de **printf** descreve os formatos nos quais os valores de saída aparecem. A string de controle de formato consiste em especificadores de conversão, sinalizadores (flags), tamanhos de campos, precisões e caracteres literais.
• Os inteiros são impressos com os seguintes especificadores de conversão: **d** ou **i** para inteiros com sinal, **o** para inteiros sem sinal na forma octal, **u** para inteiros sem sinal na forma decimal e **x** ou **X** para inteiros sem sinal na forma hexadecimal. O modificador **h** ou **1** é prefixado aos especificadores de conversão anteriores para indicar um inteiro **short** ou **long,** respectivamente.
• Os valores de ponto flutuante são impressos com os seguintes especificadores de conversão: **e** ou **E** para a notação exponencial, **f** para a notação regular de ponto flutuante e **g** ou **G** tanto para a notação **e** (ou **E)** como **f**. Quando o especificador de conversão **g** (ou **G)** é indicado, o especificador de conversão **e** (ou **E)** é usado se o expoente do valor for menor do que **-4** ou maior ou igual à precisão com a qual o valor é impresso.
• A precisão para os especificadores **g** ou **G** indica o número máximo de dígitos significativos impressos. • O especificador de conversão **c** imprime um caractere.
• O especificador de conversão **s** imprime uma string de caracteres terminando com o caractere nulo **(NULL).**
• O especificador de conversão **p** exibe um endereço de ponteiro de uma forma definida pela implementação (em muitos sistemas, é usada a notação hexadecimal).
• O especificador de conversão **n** armazena o número de caracteres já enviados ao dispositivo de saída na instrução **printf** atual. O argumento correspondente é um ponteiro para um inteiro.
• A especificação de conversão %% faz com que seja enviado um caractere literal % para o dispositivo de saída.
• Se o tamanho do campo for maior do que o objeto a ser impresso, esse último é normalmente justificado à direita no campo.
• Os tamanhos de campos podem ser usados com todos os especificadores de conversão.
• **A** precisão usada com especificadores de conversão de inteiros indica o número mínimo de dígitos impressos. Se o valor possuir menos dígitos do que a precisão especificada, são colocados zeros à frente do valor impresso até que o número de dígitos seja equivalente à precisão.
• A precisão usada com especificadores de conversão de pontos flutuantes **e, E** e **f** indica o número de dígitos que aparece depois do ponto decimal.
• A precisão usada com especificadores de conversão de pontos flutuantes **g** e G indica o número de dígitos significativos que deve aparecer.
• A precisão usada com o especificador de conversão **s** indica o número de caracteres a ser impresso

• O tamanho do campo e a precisão podem ser combinados colocando o tamanho do campo seguido de um ponto decimal e colocando a seguir a precisão entre o sinal de porcentagem e o especificador de conversão.
• E possível especificar o tamanho do campo e a precisão por meio de expressões inteiras na lista de argumentos, após a string de controle do formato. Para usar esse recurso, insira um * (asterisco) no lugar do tamanho do campo ou da precisão. O argumento correspondente na lista de argumentos do calculado e usado no lugar do asterisco. O valor do argumento pode ser negativo para o tamanho de campo mas deve ser positivo para a precisão.
• O sinalizador - justifica seu argumento à esquerda em um campo.
• O sinalizador + imprime um sinal de adição para valores positivos e um sinal de subtração para valores negativos.
• O sinalizador de espaço imprime um espaço precedendo um valor positivo não exibido com o sinalizador +.
• O sinalizador # coloca 0 à frente de valores octais, 0**x** ou 0**X** à frente de valores hexadecimais e obriga a impressão do ponto decimal em valores de ponto flutuante impressos com e, **E, f, g ou E** (normalmente o ponto decimal só é exibido se o valor possuir uma parte fracionária).
• O sinalizador 0 imprime zeros iniciais em um valor que não ocupa completamente o tamanho de campo.
• A formatação exata dos dados de entrada é realizada com a função **scanf** da biblioteca.
• Os inteiros são fornecidos como dados de entrada com os especificadores d e i para inteiros com sinal e **o, u, x** ou **X** para inteiros sem sinal. Os modificadores h e 1 são colocados antes de um especificador de conversão de inteiro para receber dados inteiros **short** e **long, respecti**vamente.
• Os valores de ponto flutuante são fornecidos como dados de entrada com os especificadores de conversão **e,** E, **f, g** ou G. Os modificadores **1** e L são colocados antes de qualquer um dos especificadores de ponto flutuante para indicar que o valor de entrada é **double** ou **long double,** respectivamente.
• Os caracteres são fornecidos como dados de entrada com o especificador de conversão c.
• As strings são fornecidas como dados de entrada com o especificador de conversão s.
• Um conjunto de varredura lê os caracteres dos dados de entrada procurando por aqueles caracteres ***que*** sejam iguais aos que fazem parte do conjunto. Quando um caractere é encontrado, é armazenado um array de caracteres. O conjunto de varredura cessa a leitura de caracteres quando encontrar um caractere que não faça parte do conjunto.
• Para criar um conjunto de varredura invertido, coloque um circunflexo (^) entre os colchetes, antes dos caracteres de varredura. Isso faz com que os caracteres que não façam parte do conjunto de varredura sejam armazenados até que um caractere contido no conjunto de varredura invertido seja encontrado.
• Os valores de endereços são fornecidos como dados de entrada com o especificador de conversão **p.**
• O especificador de conversão **n** armazena o número de caracteres recebidos anteriormente como dados de entrada no **scanf** atual. O argumento correspondente é um ponteiro a **int.**
• A especificação de conversão %% coloca um único caractere ***%*** na entrada de dados.
• O caractere de supressão de atribuição e usado para ler dados do fluxo de entrada e ignorar dados.
• O tamanho do campo é usado em **scanf** para ler um número específico de caracteres do fluxo de entrada.

Terminologia

* em precisão
* tamanho de campo
<stdio.h> alhinamento arredondamento
caractere de supressão de atribuição (*)
Caracteres literais Circunflexo (^)
Conjunto de varredura
Conjunto de varredura invertido
espaço em branco
especificação de conversão
especificador dc conversão %
especificador de conversão **c**
especificador de conversão d
especificador de conversão **e** ou **E**
especificador de conversão **f**
especificador de conversão *g* ou G
especificador de conversão h
especificador de conversão **i**
especificador de conversão **L**
especificador de conversão **1**
especificador dc conversão n
especificador de conversão **o**
especificador de conversão **p**
especificador de conversão **s**
especificador dc conversão **u**
especificador de conversão **x** (ou **X)**
especificadores de conversão de inteiros
fluxo (stream)
fluxo de entrada padrão
fluxo de erro padrão
fluxo de saída padrão
formato exponencial de ponto flutuante
formato hexadecimal
formato inteiro com sinal
formato inteiro sem sinal
formato octal
inserção de branco (espaço)
inserção de caractere imprimível
inteiro **long**
inteiro **short**
justificação à direita
justificação à esquerda
notação científica
ponto flutuante
precisão
printf
redirecionar um fluxo **scanf**
seqüência de escape
seqüência de escape \'
seqüência de escape \"
seqüência de escape \?
seqüência de escape \ \
seqüência de escape **\a**
seqüência de escape **\b**
seqüência de escape **\f**
seqüência de escape **\n**
seqüência de escape **\r**
seqüência de escape **\t**
seqüência de escape **\v**
sinalizador sinalizador (flag) #
sinalizador + (sinal de adição)
sinalizador - (sinal de subtração)
sinalizador **0** (zero)
sinalizador de espaço
string de controle de formato
tamanho (largura) de campo

Erros Comuns de Programação

9.1 Esquecer de colocar a string de controle de formato entre aspas duplas.

9.2 Imprimir um valor negativo com um especificador de conversão que aguarda um valor sem sinal (unsigned)

9.3 Usar *%c* para imprimir o primeiro caractere de uma string. A especificação de conversão *%c* exige um argumento **char.** Uma string é um ponteiro para **char,** i.e., um **char *.**

9.4 Usar *%s* para imprimir um argumento **char.** A especificação de conversão *%s* exige um argumento do tipo ponteiro para **char.** Em alguns sistemas, isso causa um erro fatal em tempo de execução chamado violação de acesso.

9.5 Usar aspas simples em torno de strings de caracteres é um erro de sintaxe. As strings de caracteres devem ser colocadas entre aspas duplas.

9.6 Usar aspas duplas em torno de uma constante de caractere. Isso cria realmente uma string constituída de dois caracteres, sendo o segundo deles o caractere **NULL** de terminação. Uma constante de caracteres é um único caractere entre aspas simples.

9.7 Tentar imprimir um caractere de porcentagem usando % em vez de %% na string de controle de formato. Quando as aparece em uma string de controle de formato, um especificador de conversão deve vir em seguida.

9.8 Não fornecer um tamanho de campo suficientemente grande para manipular o valor a ser impresso. Isso pode compensar a impressão de outros dados e pode produzir saídas confusas. Conheça seus dados!

9.9 Tentar imprimir aspas simples, aspas duplas, um sinal de interrogação ou uma barra invertida como dado literal em uma instrução **printf** sem preceder aquele caractere de uma barra invertida para formar a seqüência de escape apropriada.

Práticas Recomendáveis de Programação

9.1 Edite concisamente as saídas para criar apresentações. Isso torna as saídas do programa mais legíveis e diminui os erros dos usuários.

9.2 Ao enviar dados para o dispositivo de saída, certifique-se de que o usuário está ciente das situações **nas quais** os dados podem estar imprecisos devido à formatação (e.g., erros de arredondamento de precisões **especifi**cas).

9.3 Durante a entrada de dados, peça ao usuário um item de dados ou alguns itens de dados por vez. Evite pedir ao usuário para entrar com muitos itens de dados em resposta a um único pedido.

Dica de Portabilidade

9.1 O especificador de conversão **p** exibe um endereço de ponteiro de uma forma dependente da implementação (em muitos sistemas a notação hexadecimal é utilizada em vez da notação decimal).

Exercícios de Revisão

9.1 Preencha as lacunas de cada uma das afirmações a seguir:

a) Toda entrada e saída é realizada na forma de _______________.

b) O fluxo de _______________ está associado normalmente ao teclado.

c) O fluxo de _______________ está associado normalmente à tela do computador.

d) A formatação precisa da saída é realizada com a função _______________.

e) A string de controle de formato pode conter_______________ , _______________. _______________, _______________ e _______________.

f) O especificador de conversão _____ ou _____ pode ser usado para enviar ao dispositivo de saída um inteirodecimal com sinal.

g) Os especificadores de conversão _____, ______ e _____ são usados para exibir inteiros sem sinal na forma octal, decimal e hexadecimal, respectivamente.

h) Os modificadores _____ e ______ são colocados antes dos especificadores de conversão de inteiros para indicar que devem ser exibidos valores inteiros **short** ou **long.**

i) O especificador de conversão ______ é usado para exibir um valor de ponto flutuante na notação exponencial

j) O modificador ________ é colocado antes de qualquer especificador de ponto flutuante para indicar que deve ser exibido um valor **long double.**

k) Os especificadores de conversão **e, E** e **f** são exibidos com _______dígitos de precisão à direita do ponto decimal se não for especificada nenhuma precisão.

l) Os especificadores de conversão ____ e _____ são usados para imprimir strings e caracteres, respectivamente.

m) Todas as strings terminam no caractere ________.

n) O tamanho de campo e a precisão em uma especificação de conversão podem ser controlados por expressões inteiras colocando um _______ em substituição ao tamanho de campo ou à precisão e colocando
uma expressão inteira no argumento correspondente da lista de argumentos.

o) O sinalizador ___ faz com que a saída seja justificada à esquerda em um campo.

p) O sinalizador ______ faz com que os valores sejam exibidos ou com o sinal de adição ou com o sinal de
subtração.

q) A formatação precisa dos dados de entrada é conseguida com a utilização da função _____________.

r) Um _______ é usado para procurar determinados caracteres em uma string e armazená-los em um array.

s) O especificador de conversão ________ pode ser usado na entrada de inteiros octais, decimais e hexadecimais com sinal.

t) O especificador de conversão ___________ pode ser usado na entrada de um valor **double.**

u) O ________ é usado para ler dados de um fluxo de entrada e ignorá-los sem atribuí-los a uma variável.

v) Um __________ pode ser usado em uma especificação de conversão **scanf** para indicar que um número específico de caracteres ou dígitos deve ser lido no fluxo de entrada.

9.2 Encontre o erro em cada uma das instruções a seguir e explique como ele pode ser corrigido.

a) A instrução seguinte deve imprimir o caractere ' **c** '
printf("%s\n", 'c');
b) A instrução seguinte deve imprimir **9.375%.**
printf("%.3f%", 9.375);
c) A instrução seguinte deve imprimir o primeiro caractere da string **"Segunda"**
printf("%c\n", "Segunda");
d) **printf(" "Uma string entre aspas" ");**
e) **printf(***%d%d,* **12, 2 0);**
f) **printf("%c", "x");**
g) **printf("%s\n", 'Ricardo');**

9.3 Escreva uma instrução para cada um dos pedidos a seguir:

a) Imprimir **1234** justificado à direita em um campo de **10** dígitos.
b) Imprimir **123.456789** na notação exponencial com um sinal (+ ou -) e **3** dígitos de precisão.
c) Ler um valor **double** na variável **numero.**
d) Imprimir **100** na forma octal precedido por **0.**
e) Ler uma string no array de caracteres **string.**
f) Ler caracteres no array **n** até que seja encontrado um caractere que não seja dígito (não-dígito).
g) Usar as variáveis inteiras **x** e **y** para especificar o tamanho do campo e a precisão usada para exibir o valor **87.4573** do tipo **double.**
h) Ler um valor da forma **3.*5%.*** Armazene a porcentagem na variável **porcento** do tipo **float** e elimine o % do fluxo de entrada. Não use o caractere de supressão de atribuição.
i) Imprimir **3.333333** como um valor **long double** com um sinal (+ ou -) em um campo de **20** caracteres com precisão **3.**

Respostas dos Exercícios de Revisão

9.1 a) Fluxos (streams). b) Entrada padrão, c) Saída padrão, d) **printf.** e) Especificadores de conversão, sinalizadores (flags), tamanhos de campo, precisões e caracteres literais, f) **d, i.** g) **o, u, x** (ou **X).** h) **h, 1.** i) **e** (ou **E).** j) **L,** k) 6.1) **s, c.** m) **NULL (' \ 0 ')■** n) asterisco (*). o) - (subtração), p) + (adição), q) **scanf.** r) Conjunto de varredura, s) **i.** t) l**e, lE, lf, lg** ou l**G.** u) Caractere de supressão de atribuição (*). v) Tamanho de campo.

9.2 a) Erro: O especificador de conversão **s** exige um argumento do tipo ponteiro para **char.**

Correção: Para imprimir o caractere ' **c** ', use a especificação de conversão *%c* ou mude ' **c** ' para **"c".**

b) Erro: Tentar imprimir o caractere literal % sem usar a especificação de conversão %%. Correção: Use %% para imprimir o caractere literal %.

c) Erro: O especificador de conversão **c** exige um argumento do tipo **char.**

Correção: Para imprimir o primeiro caractere de " **Segunda"** use a especificação de conversão **%1s.**

d) Erro: Tentar imprimir o caractere literal " sem usar a seqüência de escape \ ". Correção: Substitua cada uma das aspas no conjunto interno de aspas por \ ".

e) Erro: A string de controle do formato não está entre aspas duplas. Correção: Coloque **%d%d** entre aspas duplas.

f) Erro: O caractere **x** está entre aspas duplas.

Correção: As constantes de caracteres a serem impressas com *%c* devem ser colocadas entre aspas simples.

h) Erro: A string a ser impressa está entre aspas simples.

Correção: Use aspas duplas em vez das aspas simples para representar uma string.

9.3 a) **printf ("%10d\n", 1234);**

b) **printf("%+.3e\n", 123.456789);**

c) **scanf(*"%1f*", &número);**

d) **printf("%#o\n", 100);**

e) **scanf("%s", string);**

0 scanf ("%[^0123456789] ", n) ;

g) **printf("%*.*f\n", x, y, 87.4573);**

h) **scanf("%f%%", &porcento);**

i) **printf("%+20.3Lf\n", 3.333333);**

Exercícios

9.4 Escreva uma instrução **printf** ou **scanf** para cada um dos pedidos a seguir:

a) Imprimir o inteiro sem sinal **40000** justificado à esquerda, com **8** dígitos, em um campo de **15 dígitos**

b) Ler um valor hexadecimal na variável **hex.**

c) Imprim1ir **200** com e sem sinal.

d) Imprimir **100** na forma hexadecimal, precedido por 0**x.**

e) Ler caracteres em um array **s** até que a letra **p** seja encontrada.

f) Imprimir **1.234** em um campo de **9** dígitos, precedido por zeros.

g) Ler um horário na forma **hh:mm:ss** armazenando as partes do horário nas variáveis inteiras horras **minuto** e **segundo.** Ignore os dois pontos (:) no fluxo de entrada. Use o caractere de supressão de atribuição.

h) Ler do dispositivo de entrada padrão uma string da forma " **caracteres**". Armazenar a string no array de caracteres **s.** Elimine as aspas do fluxo de entrada.

i) Ler um horário no formato **hh:mm:ss** armazenando as partes do horário nas variáveis inteiras hora **minuto** e **segundo.** Ignore os dois pontos no fluxo de entrada. Não use o caractere de supressão de atribuição.

9.5 Mostre o que será impresso em cada uma das instruções que se seguem. Se uma instrução estiver incorreta, indique por quê.

a) **printf("%-10d\n", 10000);**

b) **printf(***"%c\n"***,   "Isso eh uma string");**

c) **printf("%*.*lf\n", 8, 3, 1024.987654);**

d) **printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);**

e) **printf("% ld\n%+ld\n", 1000000, 1000000);**

f) **printf("%10.2E\n", 444.93738);**

g) **printf("%10.2g\n", 444.93738);**

h) **printf("%d\n", 10.987);**

9.6 Encontre o(s) erro(s) em cada um dos seguintes segmentos de programas. Explique como cada erro pode ser corrigido.

a) **printf("%s\n",   'Feliz Aniversário');**

b) **printf("%c\n",   'Hello') ;**

c) **printf("%c\n",   "Isso eh uma string");**

d) A instrução a seguir deve imprimir " **Bon Voyage".**

printf(" "%s" ",   "Bon Voyage");

e) **char dia[] = "Segunda"; printf("%s\n", dia[3]);**

f) **printf ('Digite seu nome: ') ;**

g) **printf(%f, 123.456);**

h) A instrução a seguir deve imprimir os caracteres '**O**' e '**K**'.

printf(*"%s%s\n"***,   'O', 'K');**

i) **char s [10] ; scanf("%c", s[7]);**

9.7 Escreva um programa que carregue um array **numero** de 10 elementos com números inteiros aleátorios de 1 a 1000. Para cada elemento, imprima o valor e um total atualizado do número de caracteres impressos. Use a especificação de conversão *%n* para determinar o número de caracteres de cada valor enviados à saida

Imprima o número total de caracteres enviados à saída por todos os valores até o valor atual, inclusive, cada vez que esse for impresso. A saída deve ter o seguinte formato:

Valor	Total de caracteres
342	3
1000	7
963	**10**
6	**11**
etc.	

9.8 Escreva um programa para examinar a diferença entre os especificadores de conversão **%d** e ***%L*** quando utilizados em instruções **scanf.** Use as instruções

scanf(*"%i%d",* **&x, &y); printf("%d %d\n", x, y) ;**

para entrar e imprimir os valores. Teste o programa com os seguintes conjuntos de dados de entrada:

10 10 -10 -10 010 010 0x10 0x10

9.9 Escreva um programa que imprima os valores de ponteiros usando todos os especificadores de conversão de inteiros e a especificação de conversão *%p.* Que especificadores imprimem valores estranhos? Que especificadores causam erros? Em que formato a especificação de conversão **%p** exibe os endereços em seu sistema?

9.10 Escreva um programa para testar os resultados da impressão do valor inteiro **12345** e o valor de ponto flutuante **1.2345** em vários tamanhos de campos. O que acontece quando os valores são impressos em campos que contêm menos dígitos que os valores?

9.11 Escreva um programa que imprima o valor **100.453627** arredondado para o dígito, décimo, centésimo, milésimo e décimo de milésimo mais próximo.

9.12 Escreva um programa que receba a entrada de uma string a partir do teclado e determine o comprimento da string. Imprima a string usando o dobro do comprimento da string como tamanho de campo.

9.13 Escreva um programa que converta temperaturas inteiras em Fahrenheit de **0** a **212** graus para temperaturas Celsius em ponto flutuante com **3** dígitos de precisão. Use a fórmula

celsius = 5.0 / 9.0 * (fahrenheit - 32);

para realizar os cálculos. A saída deve ser impressa em duas colunas justificadas à direita com 10 caracteres cada e as temperaturas Celsius devem ser precedidas de um sinal tanto para valores positivos quanto negativos.

9.14 Escreva um programa para testar todas as seqüências de escape da Fig. 9.16. Para as seqüências de escape que movem o cursor, imprima um caractere antes e depois de imprimir a seqüência de modo que fique claro para onde o cursor se moveu.

9.15 Escreva um programa que determine se ? pode ser impresso como parte de uma string de controle de formato de **printf** como um caractere literal em vez de usar a seqüência de escape \?.

9.16 Escreva um programa que receba a entrada do valor **437** usando cada um dos especificadores de conversão de inteiros de **scanf.** Imprima cada valor de entrada usando

todos os especificadores de conversão de inteiros.

9.17 Escreva um programa que use cada um dos especificadores de conversão **e, f** e **g** para receber a entrada do valor **1.2345.** Imprima os valores de cada variável para provar que cada especificador de conversão pode ser usado para receber a entrada do mesmo valor.

9.18 Em algumas linguagens de programação, as strings são fornecidas entre aspas simples *ou* duplas. Escreva um programa que leia três strings **suzy, "suzy**" e '**suzy**'. As aspas simples e duplas são ignoradas pelo C ou lidas como parte da string?

9.19 Escreva um programa que determine se ? pode ser impresso com a constante de caractere '?' em vez da seqüência de escape da constante de caractere '\?' usando o especificador de conversão *%c* na string de controle de formato de uma instrução **printf.**

9.20 Escreva um programa que use o especificador de conversão **g** para enviar à saída o valor **9876.12345.**
Imprima o valor com precisões variando de **1** a **9.**

10

Estruturas, Uniões, Manipulações de Bits e Enumerações

Objetivos

• Ser capaz de criar e usar estruturas, uniões e enumerações.
• Ser capaz de passar estruturas a funções por meio de chamadas por valor e por referência.
• Ser capaz de trabalhar com dados utilizando operadores de manipulação de bits.
• Ser capaz de criar campos de bits para armazenar dados de forma compactada

Nunca entendi o que significam esses malditos pontos.
Winston Churchill

Mas ainda uma união dividida;
William Shakespeare

Não me inclua nisso.
Samuel Goldwyn

A mesma mentira velha e generosa
Repetida ao longo dos anos
Sempre faz muito sucesso — "Você realmente não mudou nada!"
Margaret Fishback

Sumário

10.1 Introdução

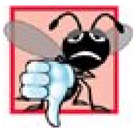
Estruturas são grupos de variáveis relacionadas entre si — algumas vezes chamadas *agregadas* — sob um nome. As estruturas podem conter variáveis de muitos tipos diferentes de dados — diferentemente de arrays que contêm apenas elementos do mesmo tipo. As estruturas são usadas normalmente para definir registros a serem armazenados em arquivos (veja o Capítulo 11, "Processamento de Arquivos"). Os ponteiros e as estruturas facilitam a formação de estruturas mais complexas de dados como listas encadeadas, filas (queues), pilhas (stacks) e árvores (veja o Capítulo 12, "Estruturas de Dados").

10.2 Definições de Estruturas

As estruturas são *tipos derivados de dados* — são construídas usando objetos de outros tipos. Examine a seguinte definição de estrutura:

```
struct carta {
        char *face;
        char *naipe;
};
```

A palavra-chave struct apresenta a definição da estrutura. O identificador carta é o *tag (marca ou rótulo) da estrutura. O* tag da estrutura dá o nome da definição da estrutura e é usado com a palavra-chave struct para declarar as variáveis do *tipo da estrutura.* Nesse exemplo, o tipo da estrutura é struct carta. As variáveis declaradas entre as chaves da definição da estrutura são os *membros* da estrutura. Os membros de uma mesma estrutura devem ter nomes exclusivos, mas duas estruturas difentes podem ter membros com mesmo nome, sem que haja conflito (veremos em breve o motivo disso). Cada definição de estrutura deve terminar com ponto-e-vírgula.



Erro comun de programação 10.1

Esquecer de colocar o ponto-e-vírgula ao terminar uma definição de estrutura.

Á definição de struct carta contém dois membros do tipo char * — face e naipe. Os membros das estruturas podem ser variáveis dos tipos básicos de dados (e.g., int, float, etc.) ou agregadas, como arrays e outras estruturas. Como vimos no Capítulo 6, todos os elementos de um array devem ser do mesmo tipo. Entretanto, os membros das estruturas podem ser de vários tipos de dado. Por exemplo, uma **struct empregado** pode conter membros strings de caracteres para o primeiro e último nomes, e um membro **int** para a idade do empregado, um membro **char** contendo 'M' ou 'F' para o sexo do empregado, um membro **float** para o valor do salário por hora trabalhada e assim por diante. Uma estrutura não pode conter uma instância de si mesma. Por exemplo, uma variavél de tipo **struct carta** não pode ser declarada na definição de **struct carta.** Entretanto, um ponteiro para **struct carta** pode ser incluído. Uma estrutura que contém um membro que é ponteiro para o mesmo tipo de estrutura é chamada *estrutura auto-referenciada.* As estruturas auto-referenciadas são usadas no Capítulo 12 para construir vários tipos de estruturas encadeadas de dados.

A definição de estrutura anterior não reserva espaço algum da memória, em vez disso a definição cria um novo tipo de dado que é usado para declarar variáveis. As variáveis de estruturas são declaradas como variáveis de outros tipos. A declaração

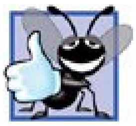
```
struct carta a, baralho[52] *cPtr;
```

declara a variável **a** como sendo do tipo **struct carta,** declara **baralho** como sendo um array com 52 elementos do tipo **struct carta** e declara **cPtr** como sendo um ponteiro para struct **carta.** As variáveis de um determinado tipo de estrutura também podem ser declaradas colocando uma lista de nomes de variáveis separados por vírgulas, entre a chave final da definição da estrutura e o ponto-e-vírgula que finaliza

aquela definição. Por exemplo, a declaração anterior poderia ter sido incorporada na estrutura **struct carta** como se segue:

```
struct carta {
        char *face;
        char *naipe;
        } a,  baralho[52], *cPtr;
```

O nome do tag da estrutura é opcional. Se uma definição de estrutura não possuir um nome de tag e variáveis do tipo da estrutura só podem ser declaradas na sua definição — não em uma declaração separada.



Boa prática de programação 10.1

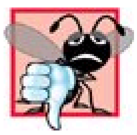
Forneça um nome de tag ao criar um tipo de estrutura. O nome do tag da estrutura mostra-se conveniente para a declaração de novas variáveis daquele tipo em um local posterior do programa.



Boa prática de programação 10.2

Escolher um nome significativo para o tag da estrutura ajuda a tornar o programa auto-explicativo.

As únicas operações válidas que podem ser realizadas em estruturas são: atribuir variaveis de estruturas a variáveis de estruturas do mesmo tipo, obter o endereço (&) de uma variável de estrurura de acesso aos membros de uma variável de estrutura (ver Seção 10.4) e usar o operador sizeof para determinar o tamanho de uma variável de estrutura.



Erro comun de programação 10.2

Atribuir uma estrutura de um tipo a uma estrutura de outro tipo.

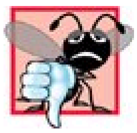
As estruturas não podem ser comparadas porque seus membros não são armazenados obrigatoriamente em bytes consecutivos da memória. Algumas vezes há "buracos" em uma estrutura porque os computadores podem armazenar tipos específicos de dados apenas em determinados limites da memória contém limites de meias-palavras, palavras e palavras duplas. Uma palavra é uma unidade-padrão de memória usada para armazenar dados em um computador — normalmente **2** ou 4 bytes. Examine a seguinte definição de estrutura na qual **amostrai** e **amostra2** do tipo **struct exemplo** são declaradas:

```
struct exemplo {
        char c;
        int i;
}amostra1, amostra2;
```

Byte	0	1	2	3
	01100001		00000000	01100001

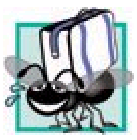
Fig. 10.1 Um alinhamento possível de armazenamento para uma variável do tipo **struct exemplo** mostrando uma área indefinida na memória.

Um computador com palavras de 2 bytes pode exigir que cada um dos membros de **struct exemplo** seja alinhado com um limite de palavra, i.e., no início de uma palavra (isso depende do equipamento utilizado). A Fig. 10.1 mostra um exemplo de alinhamento de armazenagem de uma variável do tipo **struct exemplo** que atribuiu o caractere ' **a**' ao inteiro **97** (são mostradas as representações de bits dos valores). Se os membros forem armazenados iniciando nos limites de palavras, haverá um buraco de 1 byte (byte 1 na figura) no armazenamento das variáveis do tipo **struct exemplo.** O valor no buraco de 1 byte é indefinido. Se os valores dos membros **amostrai** e **amostra2** forem realmente iguais, as estruturas não serão necessariamente consideradas iguais porque provavelmente os buracos indefinidos de 1 byte não contêm valores idênticos.



Erro comun de programação 10.3

Comparar estruturas é um erro de sintaxe devido às diferentes exigências de alinhamento em vários sistemas.



Dicas de portabilidade 10.1

Devido ao tamanho dos itens de dados de um determinado tipo ser dependente do equipamento utilizado e como as considerações de alinhamento de armazenagem também são dependentes do equipamento, da mesma forma o será a representação de uma estrutura.

10.3 Inicializando Estruturas

As estruturas podem ser inicializadas usando listas de inicializadores como arrays. Para inicializar uma etrutura, coloque, depois do nome da variável, um sinal de igual e, entre chaves, uma lista de incializadores separados por vírgulas.

struct carta a = {"Tres", "Copas"};

cria a variável **a** do tipo **struct carta** (da forma definida anteriormente) e inicializa o membro face como " **Tres**" e o membro **naipe** como " **Copas**". Se houver menos inicializadores na lista do que membros na estrutura, os membros restantes são inicializados automaticamente com 0 (ou **NULL se** o membro for um ponteiro). As variáveis de estruturas declaradas fora da definição de uma função (i.e., externamente) são inicializadas com 0 ou **NULL** se não forem inicializadas explicitamente na declaração externa. As variáveis de estruturas também podem ser inicializadas em instruções de atribuição atribuindo valores a cada um dos membros da estrutura.

10.4 Acesso a Membros de Estruturas

São usados dois operadores para acesso a membros de estruturas: O *operador de membro de estrutura (.)* — também chamado *operador de ponto* — e o *operador de ponteiro de estrutura* (->) — também chamado *operador de seta.* O operador de membro de estrutura acessa um membro de uma estrutura por meio do nome da variável da estrutura. Por exemplo, para imprimir o membro **naipe** da estrutura **a** da declaração anterior, use a instrução

printf("%s", a.naipe);

O operador de ponteiro de estrutura — que consiste em um sinal de menos (-) e de um sinal de maior que (>) sem espaços intermediários — oferece acesso a um membro de uma estrutura por meio de um ponteiro para a estrutura. Assuma que o ponteiro **aPtr** foi declarado para apontar para **struct carta** e que o endereço da estrutura **a** foi atribuído a **aPtr.** Para imprimir o membro **naipe** da estrutura **a** com o ponteiro **aPtr,** use a instrução

printf("%s", aPtr->naipe);

A expressão **aPtr->naipe** é equivalente a **(*aPtr).naipe** que desreferencia o ponteiro e tem acesso ao membro **naipe** usando o operador de membro de estrutura. Os parênteses são necessários aqui porque o operador de membro de estrutura (.) tem precedência maior do que o operador de desreferenciamento do ponteiro (*). O operador de ponteiro de estrutura e o operador de membro de estrutura, juntamente com os parênteses e colchetes ([ ]) usados para subscritos de arrays, sãc operadores que possuem a maior precedência e fazem associações da esquerda para a direita.



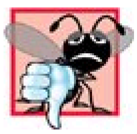
Boa prática de programação 10.3

Evite usar os mesmos nomes para membros de estruturas diferentes. Isso épermitido, mas pode causar confusão.



Boa prática de programação 10.4

Não coloque espaços em torno dos operadores -> e .. Isso ajuda a enfatizar que as expressões nas quais os operadores estão contidos são essencialmente nomes de variáveis isoladas.



Erro comun de programação 10.4

Inserir espaço entre os componentes - e >do operador de ponteiro de estrutura (ou inserir espaços entre os componentes de qualquer outro operador que necessita digitar mais de uma tecla, exceto ?:).

Erro comun de programação 10.5

Tentar fazer referência a um membro de uma estrutura usando apenas o nome do membro.

Erro comun de programação 10.6

*Não usar parênteses ao fazer referência a um membro de uma estrutura usando um ponteiro e o operador de membro de estrutura (e.g., *aPtr.naipe éum erro de sintaxe).*

O programa da Fig. 10.2 demonstra o uso dos operadores de membro de estrutura e de ponteiro de estrutura. Usando o operador de membro de estrutura, aos membros da estrutura **a** são atribuídos os valores **"As"** e " **Espadas**", respectivamente. O ponteiro **aPtr** é atribuído ao endereço da estrutura **a.** Uma instrução **printf** imprime os membros da variável de estrutura **a** usando o operador de membro de estrutura com o nome da variável **a,** o operador de ponteiro de estrutura com o ponteiro aPtr e o operador de membro de estrutura com o ponteiro **aPtr** desreferenciado.

10.5 Usando Estruturas com Funções

As estruturas podem ser passadas a funções passando cada um dos membros isoladamente, passando uma estrutura inteira ou passando um ponteiro para uma estrutura. Quando estruturas ou membro isolados são passados a uma função, eles são passados por uma chamada por valor. Portanto, os membros de uma estrutura chamadora não podem ser modificados pela função chamada.

Para passar uma estrutura chamada por referência, passe o endereço da variável da estrutura. Os arrays de estruturas — como todos os outros arrays — são passados automaticamente por meio de uma chamada por referência.

No Capítulo 6, afirmamos que um array poderia ser passado por meio de uma chamada por valor usando uma estrutura. Para passar um array por meio de uma chamada por valor, cria uma estrutura que tenha o array como um membro. Como as estruturas são passadas por meio de chamadas por valor, o array também será passado por meio de uma chamada por valor.

```
/* Usando os operadores de membro de estrutura
        e de ponteiro de estrutura */
#include <stdio.h>

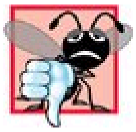
struct carta {
        char *face;
        char *naipe;
};

main(){

struct carta a;
struct carta *aPtr;
a.face = "As";
a.naipe = "Espadas";
aPtr = &a;
printf("%s%s9ós\n%s%s%s\n%s%s%s\n",a.face, "de",a.naipe, aPtr->face, "de ",
        aPtr->naipe, (*aPtr).face, "de ",(*aPtr).naipe);
return 0;
}
```

```
As de Espadas
As de Espadas
As de Espadas
```

Fig. 10.2 Usando o operador de membro de estrutura e o operador de ponteiro de estrutura.



Erro comun de programação 10.7

Admitir que estruturas, como arrays, são passados automaticamente por meio de chamada por referência e ***tentar*** *modificar os valores da estrutura chamadora na função chamada.*



Dica de desempenho 10.1

Passar estruturas por meio de chamadas por referência é mais eficiente do que passar estruturas por meio de chamadas por valor (que exige que toda a estrutura seja copiada).

10.6 Typedef

A pallavra-chave **typedef** fornece um mecanismo para a criação de sinônimos (ou aliases) para tipos de da*dos* definidos previamente. Os nomes dos tipos de estruturas são definidos freqüentemente com **typedef** para criar nomes mais curtos de tipos. Por exemplo, a instrução

typedef struct carta Carta;

define o novo nome de tipo **Carta** como um sinônimo do tipo **struct carta.** Os programadores da linuagem C usam freqüentemente **typedef** para definir um tipo de estrutura de modo que não é exigindo tag de estrutura. Por exemplo, a seguinte definição

```
typedef struct {
    char *face;
    char *naipe;
} Carta;
```

cria o tipo de estrutura **Carta** sem a necessidade de uma instrução **typedef** separada.



Boa prática de programação 10.5

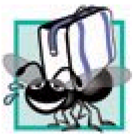
*Coloque iniciais maiúsculas nos nomes de **typedef** para enfatizar que esses nomes são sinônimos de nomes de outros tipos.*

Agora **Carta** pode ser usado para declarar variáveis do tipo **struct carta.** A declaração

Carta baralho[52];

declara um array com 52 estruturas **Carta** (i.e., variáveis do tipo **struct carta).** Criar um novo nome com **typedef** não cria um novo tipo: **typedef** simplesmente cria um novo nome de um tipo que pode ser usado como um alias (apelido) de um nome de um tipo existente. Um nome significativo ajuda a tornar o programa autodocumentado. Por exemplo, quando lemos a declaração anterior sabemos que **"baralho** é um array de 52 **Cartas".**

Freqüentemente, **typedef** é usado para criar sinônimos de tipos básicos de dados. Por exemplo um programa que exija inteiros de 4 bytes pode usar o tipo **int** em um sistema e o tipo **long e, outro.** Os programas que devem apresentar portabilidade usam freqüentemente **typedef** para criar um alias para inteiros de 4 bytes como **Integer.** O alias **Integer** pode ser modificado uma vez no programa para fazer com que ele funcione em ambos os sistemas.



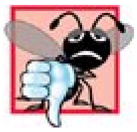
Dicas de portabilidade

*Use **typedef** para ajudar a tornar o programa mais portátil.*

10.7 Exemplo: Simulação Avançada de Embaralhamento e Distribuição de Cartas

O programa da Fig. 10.3 está baseado na simulação de embaralhamento e distribuição de cartas analisada no Capítulo 7. O programa representa um baralho de cartas como um array de estruturas. O programa usa algoritmos de alto desempenho para embaralhamento e distribuição. A saída do programa de alto desempenho para embaralhamento e distribuição de cartas é mostrada na Fig. 10.4.

No programa, a função **completaBaralho** inicializa o array **Carta,** colocando em ordem as cartas. de As a Rei, de cada naipe. O array **Carta** é passado para a função **embaralhar** onde o algoritmo de alto desempenho para embaralhamento é implementado. A função **embaralhar** usa um array de 52 estruturas **Carta** como argumento. A função faz um loop pelas 52 cartas (array com subscritos de 0 a 51) usando uma estrutura **for.** Para cada carta, é selecionado aleatoriamente um número entre 0 e 51 A seguir, a estrutura **Carta** atual e a estrutura **Carta** selecionada aleatoriamente são permutadas no array. Um total de 52 permutas é feito cm uma única passada por todo o array, c o array de estruturas **Carta** fica embaralhado! Esse algoritmo não pode sofrer de retardamento infinito como o algoritmos apresentado no Capítulo 7. Como os locais das estruturas **Carta** foram permutados no array, o algoritmo de alto desempenho para distribuição das cartas, implementado na função **distribuir,** exige apenas uma passada no array para distribuir as cartas embaralhadas.



Erro comun de programação 10.8

Esquecer-se de incluir o subscrito do array ao ser feita referência a estruturas individuais em arrays de estruturas.

10.8 Uniões

Uma ***união*** é um tipo derivado de dados — como uma estrutura — cujos membros compartilham o mesmo espaço de armazenamento. Para diferentes situações de um programa, algumas \ ariáveis podem não ser apropriadas, mas outras, são. Assim sendo, uma união compartilha o espaço em vez de desperdiçar armazenamento em variáveis que não estão sendo usadas. Os membros de uma união podem ser de qualquer tipo. O número de bytes usado para armazenar uma união deve ser pelo menos o suficiente para conter o maior membro. Na maioria dos casos, as uniões contêm dois ou mais tipos de dados. Apenas um membro, e portanto apenas um tipo de dado, pode ser referenciado de cada vez. É responsabilidade do programador assegurar que os dados de uma união sejam referenciados com o tipo apropriado.

```
/* Programa de embaralhamento e distribuição
de cartas usando estruturas */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct carta {
        char *face;
        char *naipe;
};

typedef struct carta Carta;

void completaBaralho(Carta *,  char *[],  char *[]);
void embaralhar(Carta *);
void distribuir(Carta *);

main() {

Carta baralho[52];
char *face[] = {"As","Dois","Tres","Quatro","Cinco","Seis",
                 "Sete","Oito","Nove","Dez", "Valete","Dama","Rei"};
char *naipe[]   =   {"Copas",   "Ouros",   "Paus", "Espadas"};
srand(time(NULL));

completaBaralho(baralho, face, naipe);
embaralhar(baralho);
distribuir(baralho);
return 0;
}

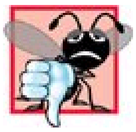
void completaBaralho(Carta *wBaralho,  char *wFace[],  char *wNaipe[]) {
int i;
for  (i = 0;   i < 52;   i++) {
        wBaralho[i].face = wFace[i % 13];
        wBaralho[i].naipe = wNaipe[i / 13];
}
```

```
}

void embaralhar(Carta *wBaralho) {
int i, j; Carta temp;
for  (i = 0;   i < 52;   i++) {
          j  = rand() % 52;
          temp = wBaralho[i];
          wBaralho[i]  = wBaralho[j];
          wBaralho[j]   = temp;
}
}

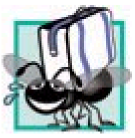
void distribuir(Carta *wBaralho) {
int i ;
for  (i = 0;   i < 52; i++)
          printf("%6s de %-7s%c", wBaralho[i].face,
                    wBaralho[i].naipe(i + 1)%2?'\.f:'\n');
}
}
```

Fig. 10.3 Simulação de embaralhamento e distribuição de cartas de alto desempenho



Erro comun de programação 10.9

Fazer referência, com o tipo errado, a dados de outro tipo armazenados em uma união é um erro lógico.



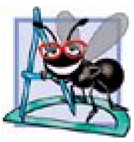
Dicas de portabilidade 10.3

Se os dados estiverem armazenados em uma união com um tipo e referenciados com outro, os resultados variam de acordo com a implementação.

Uma união é declarada com a palavra-chave **union** no mesmo formato que uma estrutura. A **decla**ração **union** indica que **numero** é um tipo de união com membros **int x** e **float y.** Normalmente, a definição **da** união precede **main** em um programa, portanto a definição pode ser usada para declarar variáveis **em** todas as funções do programa.

```
Oito de Ouros          Seis de Copas
As de Copas            Três de Espadas
Oito de Paus           Nove de Ouros
Cinco de Espadas       As de Ouros
Sete de Copas          Valete de Espadas
Dois de Ouros          Cinco de Paus
As de Paus             Rei de Ouros
Dez de Ouros           Sete de Paus
Dois de Espadas        Nove de Espadas
Seis de Ouros          Quatro de Copas
Sete de Espadas        Seis de Espadas
Dois de Paus           Oito de Espadas
Valete de Paus         Dama de Ouros
Dez de Espadas         Cinco de Ouros
Rei de Copas           As de Espadas
Valete de Ouros        Nove de Copas
Três de Copas          Rei de Paus
Três de Ouros          Cinco de Copas
Três de Paus           Rei de Espadas
Nove de Paus           Quatro de Ouros
Dez de Copas           Dama de Copas
Dois de Copas          Oito de Copas
Dez de Paus            Quatro de Espadas
Sete de Ouros          Valete de Copas
Seis de Paus           Quatro de Paus
Dama de Espadas        Dama de Paus
```

Fig. 10.4 Saída da simulação de alto desempenho para embaralhamento e distribuição de cartas.



Observação de engenharia de software 10.1

Da mesma forma que uma declaração ***struct,*** *uma declaração* **union** *simplesmente cria um novo tipo. Colocar uma declaração* **union** *ou* ***struct*** *fora de qualquer função não cria uma variável global. .*

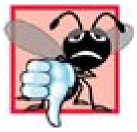
As operações que podem ser realizadas em uma união são: atribuir uma união a outra união do mes*mo* tipo, obter o endereço (&) de uma união e ter acesso aos membros de uma união usando o operador de membro de estrutura e o operador de ponteiro de estrutura. As uniões não podem ser comparadas pelas mesmas razões que levam à comparação de estruturas não ser possível.

Em uma declaração, uma união pode ser inicializada apenas com um valor do mesmo tipo que o do primeiro membro da união. Por exemplo, com a união precedente, a declaração

union numero valor = {10};

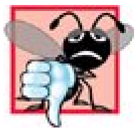
é uma inicialização válida da variável de união **valor** porque a união é inicializada com um **int,** mas a declaração a seguir seria inválida:

union numero valor = {1.43};



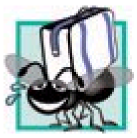
Erro comun de programação 10.10

Comparar uniões é um erro de sintaxe devido às diferentes exigências de alinhamento em vários sistemas.



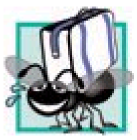
Erro comun de programação 10.11

Inicializar uma união em uma declaração com um valor cujo tipo é diferente do tipo do primeiro membro daquela união.



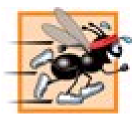
Dicas de portabilidade 10.4

A quantidade de armazenamento exigida para armazenar uma união varia de acordo com a implementação.



Dicas de portabilidade 10.5

Algumas uniões não podem ser transportadas facilmente para outros sistemas computacionais. Se uma união é portátil ou não depende das exigências de alinhamento de armazenagem dos tipos de dados dos membros da união em um detenninado sistema.



Dica de desempenho 10.2

As uniões conservam o armazenamento.

O programa da Fig. 10.5 usa a variável **valor** do tipo **union numero** para exibir o valor armazenado na união ou como **int** ou como **float.** A saída do programa varia de acordo com a implemen-tação. A saída do programa mostra que a representação interna de um valor **float** pode ser muito diferente da representação de **int.**

10.9 Operadores de Manipulação de Bits

Todos os dados são representados internamente por computadores como seqüências de bits. Cada bit pode de assumir o valor **0** ou **1.** Na maioria dos sistemas, uma seqüência de 8 bits forma um byte — a unidade-padrão de armazenamento para uma variável do tipo **char.** Outros tipos de dados são armazenados em números maiores de bytes. Os operadores de manipulação de bits (bitwise) são usados para manipular os bits de operandos integrais **(char, short, int** e **long);** tanto **signed** quanto **unsigned).** Os inteiros sem sinal **(unsigned)** são usados normalmente com os operadores de bits.

```
/* Um exemplo de união */
#include <stdio.h>

union numero {
        int x;
        float y;
};

main() {

union numero valor;
valor.x = 100;

printf("%s\n%s\n%s%d\n%s%f\n\n",
        "Coloque um valor no membro inteiro",
        "e imprima ambos os membros.",
        "int: ",  valor.x,
        "float:    ",  valor.y);

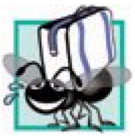
valor.y = 100.0;

printf("%s\n%s\n%s%d\n%s%f\n",
        "Coloque um valor no membro de ponto flutuante",
        "e imprima ambos os membros.",
        "int:      ", valor.x,
        "float:    ", valor.y); return 0;
}
```

```
Coloque um valor no membro inteiro
e imprima ambos os membros.
int: 100
float: 0.000000

Coloque um valor no membro de ponto flutuante
e imprima ambos os membros.
int: 17096
float: 100.000000
```

Fig. 10.5 Imprimindo o valor de uma união em ambos os tipos de dados dos membros.



Dicas de portabilidade 10.6

As manipulações de dados na forma de bits variam de acordo com o equipamento.

Observe que as análises de operadores de manipulação de bits"desta seção mostram representações binárias de operandos inteiros. Para obter uma explicação detalhada do sistema binário (também chamado de base 2) de numeração veja o Apêndice D, "Sistemas de Numeração". Além disso, os programas das Seções 10.9 e 10.10 foram testados em um Apple Macintosh usando Think C e em um PC usando Borland C++. Ambos os sistemas usam inteiros de 16 bits (2 bytes). Devido à natureza dependente de equipamento das manipulações de bits, esses programas podem não funcionar em seu sistema.

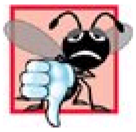
Os operadores de bits são: *E bit a bit (bitwise AND, &), OU inclusivo bit a bit (bitwise inclusive OR, ij, OU exclusivo bit a bit (bitwise exclusive OR, ^{A}), deslocamento à esquerda (leftshift, «), deslocamento de direita (right shift, »)e complemento (complement, ~).* Os operadores E bit a bit, OU inclusivo bit a bit e OU exclusivo bit a bit comparam um bit de cada um de seus dois operandos por vez. O operador E bit a bit estabelece cada bit do resultado como 1 se os bits correspondentes em ambos os operandos forem 1. O operador OU inclusivo bit a bit estabelece como 1 cada bit do resultado se o bit correspondente em um dos operandos (ou em ambos) for igual a 1. O operador OU exclusivo estabelece cada bit do resultado como 1 se o bit correspondente de exatamente um operando for igual a 1. O operador de deslocamento à esquerda faz um deslocamento dos bits de seu operando esquerdo para a esquerda, no valor do número bits especificado no operando direito. O operador de deslocamento à direita faz um deslocamento dos bits de seu operando esquerdo para a direita, no valor do número bits especificado no operando direito. O operador complemento bit a bit estabelece como **1,** no resultado, todos os bits **0** de seu operando e como **0,** no resultado, todos os bits **1.** Nos exemplos que se seguem aparece uma análise detalhada de cada operador de manipulação de bits. Os operadores de manipulação de bits estão resumidos na Fig. 10.6.

Ao usar os operadores de manipulação de bits, é útil imprimir os valores em suas representações binárias para ilustrar os efeitos exatos daqueles operadores. O programa da Fig. 10.7 imprime um inteiro **unsigned** em sua representação binaria, em grupos de 8 bits cada. A função **exibeBits** usa o operador E bit a bit para combinar a variável **valor** com a variável **exibeMascara.** Freqüentemente, o operador E bit a bit é usado com um operando chamado *máscara* — um valor inteiro com bits específicos definidos como **1.** As máscaras são usadas para ocultar alguns bits em um valor ao selecionar outros bits. Na função **exibeBits,** à variável de máscara **exibeMascara** é atribuída o valor **1 << 15 (10000000 00000000).**O operador de deslocamento à esquerda desloca o valor **1** do bit de ordem mais baixa (extremidade direita) para o bit de maior ordem (extremidade esquerda) e preenche com bits 0 a partir da direita. A instrução

putchar(valor & exibeMascara ? '1' : ' 0 ') ;

determina se um **1** ou um **0** deve ser impresso para o bit da extremidade esquerda da variável **valor,**

Assuma que a variável **valor** contém **65000 (11111101 11101000).** Quando **valor** e **exibeMascara** são combinados usando &, todos os bits, exceto o de mais alta ordem, na variável **valor** são "mascarados" (ocultos) porque qualquer bit "somado" (por meio do E) a **0** conduz a **0.** Se o bit da extremidade esquerda for **1, valor & exibeMascara** resulta em **1,** e **1** é impresso — caso contrário, 0 é impresso. A variável **valor** é então deslocada um bit pela expressão **valor <<= 1** (isso é equi**valente a valor = valor << 1).** Esses passos são repetidos para cada bit na variável **unsigned valor.** A Fig. 10.8 resume os resultados da combinação de dois bits com o operador E bit a bit.



Erro comun de programação 10.12

Usar o operador E lógico (&&) como o operador E bit a bit (&) e vice-versa.

Operador	Descrição
& E bit a bit	Os bits no resultado assumem o valor 1 se os bits correspondentes nos dois operandos forem iguais a 1.
| OU inclusivo bit a bit	Os bits no resultado assumem o valor **1** se pelo menos um dos bits correspondentes nos dois opcrandos for igual a **1.**
^ OU exclusivo bit a bit	Os bits no resultado assumem o valor **1** se exatamente um dos bits correspondentes nos dois operandos for igual a **1.**
<< deslocamento a esquerda	Desloca os bits do primeiro operando para a esquerda conforme o número de bits especificado pelo segundo operando; preenche com zeros a partir da direita.
>> deslocamento a direita	Desloca os bits do primeiro operando para a direita conforme o número de bits especificado pelo segundo operando; o método de preenchimento a partir da esquerda varia de acordo com o equipamento.
~ complemento de um	Todos os bits **0** assumem o valor **1** e todos os bits **1** assumem o valor **0.**

Fig. 10.6 Os operadores de manipulação de bits.

```
/* Imprimindo em bits um inteiro unsigned */
#include <stdio.h>

main() {

unsigned x;
void exibeBits(unsigned);
printf("Digite um inteiro unsigned: ");
scanf("%u", &x);
exibeBits(x);
return 0;
}

void exibeBits(unsigned valor) {
unsigned c,  exibeMascara = 1 << 15;
printf("%7u =  ", valor);
```

```
        for  (c = 1;  c <= 16;  c++) {
                putchar (valor & exibeMascara ? '1' : ' 0 ' );
                valor <<= 1;

                if  (c % 8 == 0)
                putchar(' ');
}
putchar('\n');
}
```

```
Digite um inteiro unsigned:
65000

65000 = 11111101 11101000
```

Fig. 10.7 Imprimindo em bits um inteiro sem sinal,

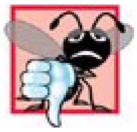
O programa da Fig. 10.9 demonstra o uso do operador E bit a bit, o operador OU inclusivo bit a bit o operador OU exclusivo bit a bit e o operador complemento bit a bit. O programa usa a função **exibeBits** para imprimir os valores inteiros **unsigned.** A saída do programa é mostrada na Fig. 10.10.

Na Fig. 10.9, à variável inteira **mascara** é atribuído o valor 1 (00000000 00000001) e à variável n**umerol** é atribuído o valor 6 5 5 3 5 (11111111 11111111). Quando **mascara** e numero1 forem combinados usando o operador E bit a bit (&) na expressão **numerol & mascara,** o resultado é **00000 00000001.** Todos os bits, exceto o bit de menor ordem na variável **numerol,** ficam "mascarados" (ocultos) pela "soma", por meio do operador E, de seus valores com os dos bits da variável **mascara.**

Bit 1	**Bit 2**	**Bit 1 & Bit 2**
0	**0**	**0**
1	**0**	**0**
0	**1**	**0**
1	**1**	**1**

Fig. 10.8 Resultados da combinação de dois bits com o operador E bit a bit &.

O operador OU inclusivo bit a bit é usado para definir bits específicos com o valor 1 em um operando. Na Fig. 10.9, à variável **numerol** é atribuído o valor **15 (00000000 00001111)** e à variável **defineBits** é atribuído o valor **241(00000000 11110001).** Quando **numerol** e **defineBits** são combinados usando o operador bit a bit OU na expressão **numerol | defineBits,** o resultado **é 255 (00000000 11111111).** A Fig. 10.11 faz um resumo dos resultados da combinação de dois bits com o operador OU inclusivo.



Erro comun de programação 10.13

*Usar o operador OU lógico (||) como o operador OU bit a bit (*I*) e vice-versa.*

O operador OU exclusivo bit a bit (^) define cada bit do resultado como 1 se *exatamente* um dos bits correspondentes em seus dois operandos for igual a 1. Na Fig. 10.9, às variáveis **numerol** e **numero2** são atribuídos os valores **139 (00000000 10001011)** e **199 (00000000 11000111),** respectivamente. Quando essas variáveis são combinadas com o operador OU exclusivo na expressão **numero1 ^ numero2,** o resultado é **00000000 0100110 0.** A Fig. 10.12 mostra um resumo dos resultados da combinação de dois bits com o operador OU exclusivo bit a bit.

```
/* Usando os operadores E bit a bit,
OU inclusivo bit a bit, OU exclusivo
bit a bit e complemento bit a bit */
#include <stdio.h>

void exibeBits(unsigned);

main() {

unsigned numerol,  numero2, mascara, defineBits;
numerol = 65535;
mascara = 1;

printf("0 resultado de combinar o seguinte\n");
exibeBits(numerol);
exibeBits(mascara);

printf("usando o operador E bit a bit & e\n");
exibeBits(numerol & mascara);

numerol = 15;
defineBits = 241;

printf("\nO resultado de combinar o seguinte\n");
exibeBits(numerol);
exibeBits(defineBits);

printf("usando o operador OU inclusivo bit a bit  I  e\n");
exibeBits(numerol| defineBits);
numerol = 139;
numero2 = 199;

printf("\nO resultado de combinar o seguinte\n");
exibeBits(numerol);
exibeBits(numero2);

```

```
printf("usando o operador OU exclusivo bit a bit ^ e\n");
exibeBits(numerol ^ numero2);

numerol = 21845;
printf("\nO complemento de um de\n");
exibeBits(numerol);

printf("e\n");
exibeBits (~ numerol) ;

return 0;
}

void exibeBits(unsigned valor){

unsigned c, exibeMascara = 1 << 15;
printf("%7u =  ", valor);
for  (c = 1;  c <= 16;  c++) {
        putchar(valor & exibeMascara ?   '1'   : '0');
        valor <<= 1;

        if  (c % 8 == 0)
                putchar('   ' ) ;
}
putchar('\n');
}
```

Fig. 10.9 Usando os operadores E bit a bit, OU inclusivo bit a bit, OU exclusivo bit a bit e complemento bit a bit.

```
O resultado de combinar o seguinte
        65535   =   11111111 11111111
              1   =   00000000 00000001
usando o operador E bit a bit & e
              1   =   00000000 00000001

O resultado de combinar o seguinte
        15   =   00000000 00001111
        241   =   00000000 11110001
usando o operador OU inclusivo bit a bit I e
        255   =   00000000 11111111

O resultado de combinar o seguinte
        139   =   00000000 10001011
        199   =   00000000 11000111
usando o operador OU exclusivo bit a bit ^ e
         76   =   00000000 01001100 »
```

```
O complemento de um de
        21845   =   01010101 01010101
e
        43690   =   10101010 10101010
```

Fig. 10.10 Saída do programa da Fig. 10.9.

O operador complemento bit a bit (~) redefine todos os bits **1** de seu operando como **0** no resultado e redefine todos os bits **0** como **1** no resultado — isso também é conhecido como "tomar o *complemento de um* do valor. Na Fig. 10.9, à variável numerol é atribuído o valor **21845 (01010101**010100101). Quando a expressão ~numerol é calculada, o resultado é **(10101010 10101010).**

Bit 1	Bit 2	Bit 1 | Bit 2
0	**0**	**0**
1	**0**	**1**
0	**1**	**1**
1	**1**	**1**

Fig. 10.11 Resultados da combinação de dois bits com o operador OU inclusivo bit a bit

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	**0**	**0**
1	**0**	**1**
0	**1**	**1**
1	**1**	**0**

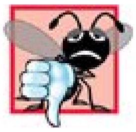
Fig. 10.12 Resultados da combinação de dois bits com o operador OU exclusivo bit a bit.

O programa da Fig. 10.13 demonstra o operador de deslocamento à esquerda (<<) e o operador de deslocamento à direita (>>). A função exibeBits é usada para imprimir valores inteiros usnsigned.

O operador de deslocamento à esquerda («) desloca os bits de seu operando esquerdo para a esquerda de acordo com o número de bits especificado em seu operando direito. Os bits liberados à direita são substituídos por Os; os ls deslocados para a esquerda, e que ficam fora do limite do operando, são perdidos. No programa da Fig. 10.13, à variável numerol é atribuído o valor **960 (0000001111000000).** O resultado de deslocar a variável numerol 8 bits para esquerda na expressão numero1 « **8** é **49152 (11000000 00000000).**

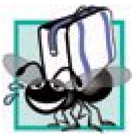
O operador de deslocamento à direita (») desloca os bits de seu operando esquerdo para a direita de acordo com o número de bits especificado em seu operando direito. Realizar um deslocamento à direita em um inteiro unsigned faz com que os bits liberados à esquerda sejam substituídos por Os; os ls deslocados para a direita, e que ficam fora do limite do operando, são perdidos. No programa da Fig. 10.13, o resultado

de deslocar à direita a variável numerol na expressão numerol >> **8** é **3 (00000000 00000011).**



Erro comun de programação 10.14

O resultado de deslocar um valor fica indefinido se o operando direito for negativo ou se o operando direito for maior do que o número de bits no qual o operando esquerdo estiver armazenado.



Dicas de portabilidade 10.7

O deslocamento à direita varia de acordo com o equipamento empregado. Deslocar à direita um inteiro com sinal preenche com Os os bits liberados em alguns equipamentos e com ls em outros.

Cada operador de manipulação de bits (exceto o operador de complemento bit a bit) tem um operador de atribuição correspondente. Esses *operadores de atribuição bit a bit* são mostrados na Fig. 10.14 e são usados de maneira similar aos operadores aritméticos de atribuição apresentados no Capítulo 3.

```
/* Usando os operadores de deslocamento de bits */
#include <stdio.h>

void exibeBits(unsigned);

main () {

unsigned numerol = 960;
printf("\nO resultado de deslocar a esquerda\n");
exibeBits(numerol);
printf("8 posições de bits usando o ");
printf("operador de deslocamento a esquerda << e\n");
exibeBits(numerol << 8);

printf("\nO resultado de deslocar a direita\n");
exibeBits(numerol);
printf("8 posições de bits usando o ");
printf("operador de deslocamento a direita >> e\n");
exibeBits(numerol >> 8);

return 0;
}

void exibeBits(unsigned valor) {

unsigned c,  exibeMascara = 1 << 15;
printf("%7u =  ", valor);

for (c = 1; c <= 16; c++) {
        putchar(valor & exibeMascara ?  '1'  : '0');
        valor <<= 1;
```

```
32.
        if  (c % 8 == 0)
            putchar('   ');
        }
        putchar('\n');
}
```

O resultado de deslocar a esquerda
    960 = 00000011 11000000
8 posições de bits usando o operador de deslocamento a esquerda << e
    49152= 11000000 00000000

O resultado de deslocar a direita
    960 = 00000011 11000000
8 posições de bits usando o operador de deslocamento a direita >> e
    3 = 00000000 00000011

Fig. 10.13 Usando os operadores de deslocamento de bits.

Operadores de atribuição bit a bit	
&=	Operador de atribuição E bit a bit
|=	Operador de atribuição OU inclusivo bit a bit
^=	Operador de atribuição OU exclusivo bit a bit
<<=	Operador de atribuição de deslocamento para esquerda
>>=	Operador de atribuição de deslocamento para direita

Fig. 10.14 Os operadores de atribuição bit a bit.

Operador	**Associatividade**	**Tipo**
( ) [ ] -> •	esquerda para a direita	Maior
+ + — + - ! ~ (tipo) * & sizeof	direita para a esquerda	Unário
* / %	esquerda para a direita	Multiplicativo
+ + -	esquerda para a direita	aditivo
« »	esquerda para a direita	Deslocamento
< <= > >=	esquerda para a direita	Relacional
== !=	esquerda para a direita	Igualdade
&	esquerda para a direita	E bit a bit
^	esquerda para a direita	Negação
|	esquerda para a direita	Ou bit a bit
&&	esquerda para a direita	E lógico
| |	esquerda para a direita	OU lógico
?:	direita para a esquerda	OU lógico Condicional
= += -= *= /= %= &= ^= |= <<= >>=	direita para a esquerda	Atribuição
`	esquerda para a direita	Vírgula

Fig. 10.15 Precedência e associatívidade dos operadores.

A Fig. **10.15** mostra a precedência e a associatívidade dos vários operadores apresentados até agora no texto. Eles são mostrados de cima para baixo na ordem decrescente de precedência.

10.10 Campos de Bits

A linguagem C fornece a capacidade de especificar o número de bits no qual um membro **unsigned** ou **int** de uma estrutura ou união é armazenado — tal especificação é chamada de *campo de bits.* Os campos de bits permitem melhor utilização da memória armazenando dados no número mínimo de bits exigido. Os membros dos campos de bits são declarados como **int** ou **unsigned.**



Dica de desempenho 10.3

Os campos de bits ajudam a conservar o armazenamento.

Examine a seguinte definição de estrutura:

```
struct bitCarta {
        unsigned face  : 4;
        unsigned naipe  : 2;
        unsigned cor : 1;
};
```

A definição contém três campos de bits **unsigned** — **face, naipe** e **cor** — usados para representar uma das **52** cartas de um baralho. Um campo de bit é declarado colocando, após o nome do membro **int** ou **unsigned,** dois pontos (:) e uma constante inteira que represente o *tamanho* do campo, i.e., o número de bits no qual o membro deve ser armazenado. A constante que representa o tamanho deve ser um inteiro entre **0** e o número total de bits usados para armazenar um valor **int** em seu sistema. Nossos exemplos foram testados em um computador com inteiros de **2** bytes **(16** bits).

A definição de estrutura mostrada anteriormente indica que o membro **face** é armazenado em **4** bits, o membro **naipe** é armazenado em **2** bits e o membro **cor** é armazenado em **1** bit. O número de bits se baseia no intervalo de valores desejado para cada membro da estrutura. O membro **face** armazena valores entre 0 (Ás) e 12 (Rei) — **4** bits podem armazenar um valor entre **0** e **15.** O membro **naipe** armazena valores entre 0 e 3 (0 = Ouros, 1 = Copas, 2 = Paus, 3 = Espadas) — **2** bits podem armazenar um valor entre 0 e 3. Finalmente, o membro **cor** armazena 0 (Vermelho) ou 1 (Preto) — **1** bit pode armazenar 0 ou 1.

O programa da Fig. 10.16 (cuja saída amostrada na Fig. 10.17) cria o array **baralho** contendo 52 estruturas **struct bitCarta.** A função **completaBaralho** insere 52 cartas no array **baralho** e a função **distribuir** imprime as 52 cartas. Observe que os membros dos campos de bits das estruturas são acessados exatamente da mesma forma que qualquer outro membro de estrutura. O membro **cor** é incluído como um meio de indicar a cor da carta em um sistema que possibilite exibições de cores.

```c
/* Exemplo usando um campo de bits */
#include <stdio.h>

struct bitCarta {
	unsigned face   : 4;
	unsigned naipe : 2;
	unsigned cor     : 1;
};

typedef struct bitCarta Carta;

void completaBaralho(Carta *);
void distribuir(Carta *);

main() {

	Carta baralho[52];
	completaBaralho(baralho);
	distribuir(baralho);
	return 0;
}

void completaBaralho(Carta *wBaralho) {
	int i;
	for  (i = 0;   i <= 51;   i++) {
		wBaralho[i].face = i % 13;
		wBaralho[i].naipe = i / 13;
		wBaralho[i].cor = i / 26;
	}
}

/* A funcao distribuir imprime as cartas no formato de duas colunas */ /* A coluna 1 contem as cartas de 0-25 com subscrito kl */ /* A coluna 2 contem as cartas de 26-51 com subscrito k2 */

void distribuir(Carta *wBaralho)
{
	int kl, k2;
	for (kl = 0,  k2 = kl + 26; kl <= 25; kl++,  k2++) {
		printf("Carta:%3d Naipe:%2d Cor:%2d ",
			wBaralho[kl].face, wBaralho[kl].naipe, wBaralho[kl].cor);
		printf("Carta:%3d Naipe:%2d Cor:%2d\n",wBaralho[k2].face,
			wBaralho[k2].naipe, wBaralho[k2].cor);
	}
}
```

Fig. 10.16 Usando campos de bits para armazenar um baralho de cartas.

```
Carta: O  Naipe: O Cor: 0 Carta: 0  Naipe: 2 Cor: 1
Carta: 1  Naipe: 0 Cor: 0 Carta: 1  Naipe: 2 Cor: 1
Carta: 2  Naipe: 0 Cor: 0 Carta: 2  Naipe: 2 Cor: 1
Carta: 3  Naipe: 0 Cor: 0 Carta: 3  Naipe: 2 Cor: 1
Carta: 4  Naipe: 0 Cor: 0 Carta: 4  Naipe: 2 Cor: 1
Carta: 5  Naipe: 0 Cor: 0 Carta: 5  Naipe: 2 Cor: 1
Carta: 6  Naipe: 0 Cor: 0 Carta: 6  Naipe: 2 Cor: 1
Carta: 7  Naipe: 0 Cor: 0 Carta: 7  Naipe: 2 Cor: 1
Carta: 8  Naipe: 0 Cor: 0 Carta: 8  Naipe: 2 Cor: 1
Carta: 9  Naipe: 0 Cor: 0 Carta: 9  Naipe: 2 Cor: 1
Carta: 10 Naipe: 0 Cor: 0 Carta: 10 Naipe: 2 Cor: 1
Carta: 11 Naipe: 0 Cor: 0 Carta: 11 Naipe: 2 Cor: 1
Carta: 12 Naipe: 0 Cor: 0 Carta: 12 Naipe: 2 Cor: 1
Carta: 0  Naipe: 1 Cor: 0 Carta: 0  Naipe: 3 Cor: 1
Carta: 1  Naipe: 1 Cor: 0 Carta: 1  Naipe: 3 Cor: 1
Carta: 2  Naipe: 1 Cor: 0 Carta: 2  Naipe: 3 Cor: 1
Carta: 3  Naipe: 1 Cor: 0 Carta: 3  Naipe: 3 Cor: 1
Carta: 4  Naipe: 1 Cor: 0 Carta: 4  Naipe: 3 Cor: 1
Carta: 5  Naipe: 1 Cor: 0 Carta: 5  Naipe: 3 Cor: 1
Carta: 6  Naipe: 1 Cor: 0 Carta: 6  Naipe: 3 Cor: 1
Carta: 7  Naipe: 1 Cor: 0 Carta: 7  Naipe: 3 Cor: 1
Carta: 8  Naipe: 1 Cor: 0 Carta: 8  Naipe: 3 Cor: 1
Carta: 9  Naipe: 1 Cor: 0 Carta: 9  Naipe: 3 Cor: 1
Carta: 10 Naipe: 1 Cor: 0 Carta: 10 Naipe: 3 Cor: 1
Carta: 11 Naipe: 1 Cor: 0 Carta: 11 Naipe: 3 Cor: 1
Carta: 12 Naipe: 1 Cor: 0 Carta: 12 Naipe: 3 Cor: 1
```

Fig. 10.17 Saída do programa da Fig. 10.16,

É possível especificar um *campo de bits anônimo* que é usado como *enchimento (padding)* na estrutura. Por exemplo, a definição de estrutura

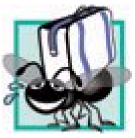
```
struct exemplo {
        unsigned a  : 13;
        unsigned  c  : 3;
        unsigned b  : 4;
        };
```

usa um campo de bits anônimo de 3 bits como enchimento — nada pode ser armazenado nesses três bits. O membro **b** (em nosso computador de palavras de 2 bytes) é armazenado em outra unidade de armazenamento.

Um *campo de bits anônimo com tamanho zero* é usado para alinhar o próximo campo de bit no limite de uma nova unidade de armazenamento. Por exemplo, a definição de estrutura

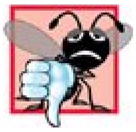
```
struct exemplo {
        unsigned a  : 13;
        unsigned  c    : 0;
        unsigned b  : 4;
        };
```

usa um campo anônimo de 0 bit para ignorar os bits restantes (tantos quantos existirem) da unidade de armazenamento na qual **a** está armazenado e alinhar **b** no limite da próxima unidade de armazenamento.



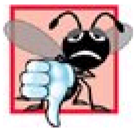
Dicas de portabilidade 10.8

As manipulações de campos de bits variam de um equipamento para outro. Por exemplo, alguns computadores permitem que os campos de bits ultrapassem os limites das palavras, ao passo que outros não.



Erro comun de programação 10.15

Tentar ter acesso a um único bit de um campo de dados como se ele fosse elemento de um array. Os campos de bits não são "arrays de bits



Erro comun de programação 10.16

Tentar obter o endereço de um campo de bits (o operador & não pode ser usado com. campos de bits porque eles não possuem endereços).



Dica de desempenho 10.4

Embora os campos de bits economizem espaço, usá-los pode fazer com que o compilador gere código ***em*** *linguagem de máquina que seja executado lentamente. Isso ocorre porque ele utiliza operações extras* ***em*** *linguagem de máquina para ter acesso apenas a porções de uma unidade de armazenamento endereçável. Isso é um dos muitos exemplos dos tipos de compensações espaço-tempo que ocorrem na ciência da computação.*

10.11 Constantes de Enumeração

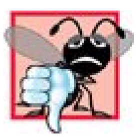
A linguagem C fornece um tipo final definido pelo usuário chamado de uma *enumeração.* Uma enumeração, apresentada pela palavra-chave **enum,** é um conjunto de constantes inteiras representadas por identificadores. Essas *constantes de enumeração* são, na realidade, constantes simbólicas cujos valores podem ser definidos automaticamente. Os valores em um **enum** iniciam com 0, a menos que seja especificado de outra forma, e são incrementados de 1. Por exemplo, a enumeração

enum meses {JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ};

cria um novo tipo, **enum meses,** no qual os identificadores são associados automaticamente aos inteiros 0 a 11. Para numerar os meses de 1 a 12, use a seguinte enumeração:

enum meses {JAN = 1, **FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ};**

Como o primeiro valor na enumeração anterior é definido explicitamente como 1, os valores restantes são incrementados de 1, resultando nos valores de 1 a 12. Os identificadores em uma enumeração devem ser exclusivos. O valor de cada constante de uma enumeração pode ser estabelecido explicitamente na definição atribuindo um valor a um identificador. Vários membros de uma enumeração podem ter o mesmo valor inteiro. No programa da Fig. 10.18, a variável de enumeração **mes** é usada em uma estrutura **for** para imprimir os meses do ano a partir do array **nomeMes.** Observe que fizemos **nomeMe[0]** possuir a string vazia " ". Alguns programadores podem preferir definir **nomeMes [** 0] com um valor como *****ERRO***** para indicar a ocorrência de um erro lógico.



Erro comun de programação 10.17

Atribuir um valor a uma constante de enumeração depois de ela ter sido definida é um erro de sintaxe.



Boa prática de programação 10.6

Use apenas letras maiúsculas em nomes de constantes de enumeração. Isso faz com que essas constantes se destaquem em um programa e lembram o programador de que as constantes de enumeração não são variáveis.

```
/* Usando um tipo de enumeração */
#include <stdio.h>

enum meses  {JAN = 1,FEV,MAR,ABR,MAI,JUN,JUL,AGO,SET,OUT,NOV,DEZ};

main(){

enum meses mes; $

char *nomeMes[]  ={"","Janeiro","Fevereiro","Marco","Abril","Maio",
            “Junho","Julho","Agosto","Setembro","Outubro","Novembro","Dezembro"};

for (mes = JAN; mes <= DEZ; mes++)
           printf("%2d%lls\n", mes,  nomeMes[mes]);

return 0;
}
```

```
1 Janeiro
2 Fevereiro
3 Marco
4 Abril
5 Maio
6 Junho
7 Julho
8 Agosto
9 Setembro
10 Outubro
11 Novembro
12 Dezembro
```

Fig. 10.18 Usando uma enumeração

Resumo

• Estruturas são conjuntos de variáveis relacionadas entre si, algumas vezes chamadas de agregradas sob um único nome.

• As estruturas podem conter variáveis de diferentes tipos de dados.

• A palavra-chave **struct** começa a definição de todas as estruturas. Dentro das chaves da definição da estrutura ficam as declarações dos membros da estrutura.

• Os membros de uma mesma estrutura devem ter nomes exclusivos.

• Uma definição de estrutura cria um novo tipo de dado que pode ser usado para declarar variáveis.

• Há dois métodos para declarar variáveis de estruturas. O primeiro é declarar as variáveis em uma declaração, da mesma forma que é feito com variáveis de outros tipos de dados usando **struct nomeTag** como tipo. O segundo método é incluir as variáveis entre a segunda chave da definição da estrutura e o ponto-e-vírgula que encerra a definição.

• O nome de tag (marca ou rótulo da estrutura) é opcional. Se a estrutura for definida em um nome de taz, as variáveis do tip derivado de dados devem ser declarados na definição da estrutura e nenhuma outra variável do novo tipo de estrutura pode ser declarada.

• Uma estrutura pode ser inicializada com uma lista de inicializadores colocando, após o nome da variável na declaração da estrutura, um sinal de igual e uma lista de inicializadores separados por vírgulas e entre chaves. Se houver menos inicializadores na lista do que membros na estrutura, os membros restantes são inicializados automaticamente com o valor zero (ou **NÜLL** se o membro for um ponteiro).

• Estruturas inteiras podem ser atribuídas a variáveis de estrutura do mesmo tipo.

• Uma variável de estrutura pode ser inicializada com uma variável de estrutura do mesmo tipo.

• Um operador de membro de estrutura é usado durante o acesso a um membro de estrutura por meio do nome da variável de estrutura.

• Um operador de ponteiro de estrutura — criado com um sinal de menos (-) e um sinal de maior que (>) — é usado no acesso a um membro de uma estrutura por meio do ponteiro para a estrutura.

• As estruturas e membros de estruturas considerados isoladamente são passados a funções por meio de chamadas por valor.

• Para passar uma estrutura por meio de uma chamada por referência, passe o endereço da variável da estrutura.

• Um array de estruturas é passado automaticamente por meio de uma chamada por referência.

• Para passar um array por meio de uma chamada por valor, crie uma estrutura com o array como menbro

• Criar um novo nome com **typedef** não cria um novo tipo; isso cria um nome que é sinônimo de um tipo definido anteriormente.

• Uma união é um tipo de dado derivado cujos membros compartilham o mesmo espaço de aarmazenamento. Os membros podem ser de qualquer tipo.

• O armazenamento reservado para uma união é suficientemente grande para armazenar seu maior número. Na maioria dos casos, as uniões contêm dois ou mais tipos de dados. Apenas um membro e portanto um tipo de dado, pode ser referenciado de cada vez.

• Uma união é declarada com a palavra-chave **union** no mesmo formato que uma estrutura.

• Uma união pode ser inicializada apenas com um valor do tipo de seu primeiro membro.

• O operador E bit a bit (bitwise AND, &) utiliza dois operandos inteiros. Um bit no resultado recebem o valor 1 se os bits correspondentes de cada operando forem 1.

• As máscaras são usadas para ocultar alguns bits e preservar outros.

• O operador OU inclusivo bit a bit (bitwise inclusive OR, I) utiliza dois operandos. Um bit do resultado receberá o valor 1 se o bit correspondente em um dos operandos for igual a 1.

• Cada um dos operadores de manipulação de bits (exceto o operador unário de complemento bit a bit tem um operador de atribuição correspondente.

• O operador OU exclusivo bit a bit (bitwise exclusive OR, ^A^) utiliza dois operandos. Um bit do resultado receberá o valor 1 se exatamente um dos bits correspondentes nos dois operandos for igual a 1

• O operador de deslocamento à esquerda (<<) desloca para a esquerda os bits de seu operando esquerdo, de acordo com o número de bits especificado por seu operando direito. Os bits liberados da direita são substituídos por Os.

• O operador de deslocamento à direita (>>) desloca para a direita os bits de seu operando esquerdo de acordo com o número de bits especificado por seu operando direito. Realizar um deslocamento à direita em um inteiro sem sinal faz com que os bits liberados da esquerda sejam substituídos por 0s Os bits liberados em inteiros com sinal podem ser substituídos por Os ou ls — isso varia com o equipamento utilizado.

• O operador de complemento bit a bit (~) utiliza um operando e inverte seus bits — isso produz o complemento de um do operando.

• Os campos de bits reduzem o uso do armazenamento armazenando dados no número mínimo de bits exigido.

• Os membros de campos de bits devem ser declarados **int** ou **unsigned.**

• Um campo de bit é declarado colocando um ponto-e-vírgula e o tamanho (comprimento) do campo após um valor **unsigned** ou **int.**

• O tamanho de um campo de bits deve ser uma constante inteira entre 0 e o número total de bits usado para armazenar uma variável **int** em seu sistema.

• Se um campo de bits for especificado sem um nome, o campo será usado como enchimento (padding) na estrutura.

• Um campo de bits anônimo com tamanho **0** é usado para alinhar o próximo campo de bits em um limite de nova palavra do equipamento.

• Uma enumeração, designada pela palavra-chave **enum,** é um conjunto de inteiros que são representados por identificadores. Os valores em um **enum** iniciam com **0** a menos que seja especificado o contrário e sempre são incrementados de **1.**

Terminologia

Refazer

Erros Comuns de Programação

10.1 Esquecer de colocar o ponto-e-vírgula ao terminar uma definição de estrutura.

10.2 Atribuir uma estrutura de um tipo a uma estrutura de outro tipo.

10.3 Comparar estruturas é um erro de sintaxe devido às diferentes exigências de alinhamento em vários sistemas.

10.4 Inserir espaço entre os componentes - e > do operador de ponteiro de estrutura (ou inserir espaços entre os componentes de qualquer outro operador que necessita digitar mais de uma tecla, exceto ? :).

10.5 Tentar fazer referência a um membro de uma estrutura usando apenas o nome do membro.

10.6 Não usar parênteses ao fazer referência a um membro de uma estrutura usando um ponteiro e o operador de membro de estrutura (e.g., ***aPtr. naipe,** é um erro de sintaxe).

10.7 Admitir que estruturas, como arrays, são passados automaticamente por meio de chamada por referência e tentar modificar os valores da estrutura chamadora na função chamada.

10.8 Esquecer-se de incluir o subscrito do array ao ser feita referência a estruturas individuais em arrays de estruturas.

10.9 Fazer referência, com o tipo errado, a dados de outro tipo armazenados em uma união é um erro lógico.

10.10 Comparar uniões é um erro de sintaxe devido às diferentes exigências de alinhamento em vários sistemas.

10.11 Inicializar uma união em uma declaração com um valor cujo tipo é diferente do tipo do primeiro membro daquela união.

10.12 Usar o operador E lógico (&&) como o operador E bit a bit (&) e vice-versa.

10.13 Usar o operador OU lógico (i I) como o operador OU bit a bit (I) e vice-versa.

10.14 O resultado de deslocar um valor fica indefinido se o operando direito for negativo ou se o operando direito for maior do que o número de bits no qual o operando esquerdo estiver armazenado.

10.15 Tentar ter acesso a um único bit de um campo de dados como se ele fosse elemento de um array. Os campos de bits não são "arrays de bits".

10.16 Tentar obter o endereço de um campo de bits (o operador & não pode ser usado com campos de bits porque eles não possuem endereços).

10.17 Atribuir um valor a uma constante de enumeração depois de ela ter sido definida é um erro de sintaxe.

Práticas Recomendáveis de Programação

10.1 Forneça um nome de tag ao criar um tipo de estrutura. O nome do tag da estrutura mostra-se conveniente para a declaração de novas variáveis daquele tipo em um local posterior do programa.

10.2 Escolher um nome significativo para o tag da estrutura ajuda a tornar o programa auto-explicativo.

10.3 Evite usar os mesmos nomes para membros de estruturas diferentes. Isso é permitido, mas pode causar confusão.

10.4 Não coloque espaços em torno dos operadores - > e .. Isso ajuda a enfatizar que as expressões nas quais os operadores estão contidos são essencialmente nomes de variáveis isoladas.

10.5 Coloque iniciais maiúsculas nos nomes de **typedef** para enfatizar que esses nomes são sinônimos de nomes de outros tipos.

10.6 Use apenas letras maiúsculas em nomes de constantes de enumeração. Isso faz com que essas constantes se destaquem em um programa e lembram o programador de que as constantes de enumeração não são variáveis.

Dicas de Portabilidade

10.1 Devido ao tamanho dos itens de dados de um determinado tipo ser dependente do equipamento utilizado e como as considerações de alinhamento de armazenagem também são dependentes do equipamentom da mesma forma o será a representação de uma estrutura.

10.2 Use **typedef** para ajudar a tomar o programa mais portátil.

10.3 Se os dados estiverem armazenados em uma união com um tipo e referenciados com outro, os resultados variam de acordo com a implementação.

10.4 A quantidade de armazenamento exigida para armazenar uma união varia de acordo com a implementação

10.5 Algumas uniões não podem ser transportadas facilmente para outros sistemas computacionais. Se uma união é portátil ou não depende das exigências de alinhamento de armazenagem dos tipos de dados dos membros da união em um determinado sistema.

10.6 As manipulações de dados na forma de bits variam de acordo com o equipamento.

10.7 O deslocamento à direita varia de acordo com o equipamento empregado. Deslocar à direita um inteiro com sinal preenche com Os os bits liberados em alguns equipamentos e com ls em outros.

10.8 As manipulações de campos de bits variam de um equipamento para outro. Por exemplo, alguns computadores permitem que os campos de bits ultrapassem os limites das palavras, ao passo que outros não.

Dicas de Performance

10.1 Passar estruturas por meio de chamadas por referência é mais eficiente do que passar estruturas por meio de chamadas por valor (que exige que toda a estrutura seja copiada).

10.2 As uniões conservam o armazenamento.

10.3 Os campos de bits ajudam a conservar o armazenamento.

10.4 Embora os campos de bits economizem espaço, usados pode fazer com que o compilador gere código em linguagem de máquina que seja executado lentamente. Isso ocorre porque ele utiliza operações extrax em linguagem de máquina para ter acesso apenas a porções de uma unidade de armazenamento endereçável. Isso é um dos muitos exemplos dos tipos de compensações espaço-tempo que ocorrem na ciência da computação.

Observação de Engenharia de Software

10.1 Da mesma forma que uma declaração **struct,** uma declaração **union** simplesmente cria um novo tipo Colocar uma declaração **union** ou **struct** fora de qualquer função não cria uma variável global.

Exercícios de Revisão

10.1 Preencha as lacunas seguintes:

a) Uma_é um conjunto de variáveis relacionadas entre si, sob um mesmo nome.

b) Uma_é um conjunto de variáveis sob um nome no qual elas compartilham o mesmo armazenamento.

c) Os bits no resultado de uma expressão usando o operador_recebem o valor 1 se os bits correspondentes em cada operando forem iguais a 1. Caso contrário, os bits recebem o valor 0.

d) As variáveis declaradas na definição de uma estrutura são chamadas suas_.

e) Os bits no resultado de uma expressão usando o operador_recebem o valor 1 se pelo menos os bits recebem um dos bits correspondentes em qualquer um dos operandos for igual a 1. Caso contrário, os bits recebem o valor zero.

f) A palavra-chave_apresenta uma declaração de estrutura.

g) A palavra-chave_é usada para criar um sinônimo de um tipo de dado definido anteriormente

h) Os bits no resultado de uma expressão que usa o operador_recebem o valor 1 se exatamente um dos bits correspondentes em qualquer dos operandos for igual a 1. Caso contrário, os bits recebem o valor zero.

i) O operador E bit a bit & é usado freqüentemente para_bits; isto é selecionar determinados bits de uma string de bits ao mesmo tempo em que oculta outros.

j) A palavra-chave_é usada para apresentar uma definição de união.

k) O nome de uma estrutura é chamado_da estrutura.

1) O membro de uma estrutura é acessado com o operador de_ou o operador de_.

m) Os operadores_e_são usados para deslocar os bits de um valor para a esquerda e para a direita, respectivamente.

n) Uma_é um conjunto de inteiros representado por identificadores.

10.2 Diga se cada uma das seguintes afirmações é verdadeira ou falsa. Se falsa, explique por quê.

a) As estruturas podem conter apenas um tipo de dado.

b) Duas uniões podem ser comparadas para determinar se são iguais.

c) O nome do tag de uma estrutura é opcional.

d) Os membros de diferentes estruturas devem ter nomes exclusivos.
e) A palavra-chave **typedef** é usada para definir novos tipos de dados.
f) As estruturas são sempre passadas a funções por meio de chamadas por referência.
g) As estruturas não podem ser comparadas.

10.3 Escreva uma instrução simples ou um conjunto de instruções para realizar cada um dos pedidos a seguir:

a) Defina uma estrutura chamada **part** contendo a variável **partNumero,** do tipo **int,** e o array **partNome,** do tipo **char,** que pode ter o comprimento de até 25 caracteres.

b) Defina **Part** como um sinônimo do tipo **struct part.**

c) Use **Part** para declarar uma variável **a** como sendo do tipo **struct part,** um array **b [10]** como sendo do tipo **struct part** e uma variável **ptr** como sendo do tipo ponteiro para **struct part.**

d) Leia um número e o nome de uma peça a partir do teclado e armazene nos membros da variável **a.**

e) Atribua os valores dos membros da variável **a** ao elemento 3 do array **b.**

f) Atribua o endereço do array **b** à variável de ponteiro **ptr.**

g) Imprima os valores dos membros do elemento 3 do array **b** usando a variável **ptr** e o operador de ponteiro de estrutura para se referir aos membros.

10.4 Encontre o erro em cada uma das instruções a seguir:

a) Assuma que **struct carta** foi definida contendo dois ponteiros de tipos **char,** que são **face e naipe.** Além disso, a variável **c** foi declarada como sendo do tipo **struct carta** e a variável **cPtr** foi declarada como sendo ponteiro para **struct carta.** O endereço de **c** foi atribuído à variável **cPtr.**

```
printf("%s\n", *cPtr->face);
```

b) Assuma que **struct carta** foi definida contendo dois ponteiros para tipos **char,** que são **face** e **naipe.** Além disso, a variável **copas [13**] foi declarada do tipo **struct carta.** A instrução a se-' guir deve imprimir o membro **face** do elemento 10 do array.

```
printf("%s\n", copas.face);
```

c)
```
union valores {
char w; float x; double y; } v = {1.27};
```

d)
```
struct pessoal {
char sobreNome [15]; char primeiroNome[15]; int idade;
}
```

e) Assuma que **struct pessoal** foi definido como na parte (d) mas com a correção adequada.

```
pessoal d;
```

f) Assuma que a variável **p** foi declarada do tipo **struct pessoal** e a variável **c** foi declarada do tipo **struct carta.**

```
p = c;
```

Respostas dos Exercícios de Revisão

10.1 a) estrutura, b) união, c) E bit a bit (&). d) membros, e) OU inclusivo bit a bit (I). f) **struct.** g) **typedef.** h) OU exclusivo bit a bit (^{A}). i) máscara, j) **union.** k) tag. 1) membro de estrutura, ponteiro de estrutura, m)operador de deslocamento à esquerda (<<), operador de deslocamento à direita (>>). n) enumeração.

10.2 a) Falso. Uma estrutura pode conter muitos tipos de dados.
b) Falso. As uniões não podem ser comparadas devido aos problemas de alinhamento associados às estruturas.
c) Verdadeiro.
d) Falso. Os membros de estruturas diferentes podem ter os mesmos nomes, mas os membros da mesma estrutura devem ter nomes diferentes.
e) Falso. A palavra-chave **typedef** é usada para definir novos nomes (sinônimos) de tipos de dados definidos anteriormente.
f) Falso. As estruturas sempre são passadas a funções por meio de chamadas por valor.
g) Verdadeiro, devido aos problemas de alinhamento.

10.3 a) **struct part {**
int partNumero; char partNome[25] ;
};
b) **typedef struct part Part;**
c) **Part a, b[10], *ptr;**
d) **scanf("%d%s", &a.partNumero, &a.partNome);**
e) **b[3] = a;**
f) **ptr = b;**
g) **printf("%d *%s\n"*, (ptr + 3)->partNumero, (ptr + 3)->partNome);**

10.4 a) Erro: Os parênteses que devem conter ***cPtr** foram omitidos, fazendo com que a ordem de cálculo da expressão seja incorreta.
b) Erro: O subscrito do array foi omitido. A expressão deve ser **copas [10] . face.**
c) Erro: Uma união só pode ser inicializada com um valor que tenha o mesmo tipo que o primeiro membro daquela união.
d) Erro: E exigido um ponto-e-vírgula para encerrar a definição da estrutura.
e) Erro: A palavra-chave **struct** foi omitida da declaração da variável.
f) Erro: Uma variável de um tipo de estrutura não pode ser atribuída a uma variável de um tipo de estrutura diferente.

Exercícios

10.5 Forneça a definição de cada uma das estruturas e uniões a seguir:
a) Estrutura **inventario** contendo o array de caracteres **partNome [3 0**], o valor inteiro **partNumero** o valor de ponto flutuante **preço,** o valor inteiro **estoque** e o valor inteiro **pedido.**
b) União **dados** contendo **char c, short s, long 1, float f edouble d.**
c) Uma estrutura chamada **endereço** que contém os arrays de caracteres **rua [25], cidade[20]estado[3]** e**codigoPostal[6].**
d) Estrutura **aluno** que contém os arrays **primeiroNome [15]** e **sobrenome [15]** e a variável **enderecoResidencia** do tipo **struct endereço** da parte (c).
e) Estrutura **teste** contendo 16 campos de bits com tamanho de 1 **bit.** Os nomes dos campos de

bits são as letras **a** a **p.**

10.6 De posse das seguintes definições de estruturas e declarações de variáveis

struct cliente <
char sobreNome[15]; char primeiroNome[15]; int numeroCliente;
struct { * char numeroFone[11];
char endereço[50];
char cidade[15];
char estado[3];
char codPostal[6]; } pessoal;
} registroCliente, *clientePtr;
clientePtr = &registroCliente;

escreva uma expressão em separado que possa ser usada para acessar membros da estrutura em cada uma das seguintes situações.

a) Membro **sobrenome** da estrutura **registroCliente.**
b) Membro **sobrenome** da estrutura apontada por **clientePtr.**
c) Membro **primeiroNome** da estrutura **registroCliente.**
d) Membro **primeiroNome** da estrutura apontada por **clientePtr.**
e) Membro **numeroCliente** da estrutura **registroCliente.**
f) Membro **numeroCliente** da estrutura apontada por **clientePtr.**
g) Membro **numeroFone** do membro **pessoal** da estrutura **registroCliente.**
h) Membro **numeroFone** do membro **pessoal** da estrutura apontada por **clientePtr.**
i) Membro **endereço** do membro **pessoal** da estrutura **registroCliente.**
j) Membro **endereço** do membro **pessoal** da estrutura apontada por **clientePtr.**
k) Membro **cidade** do membro **pessoal** da estrutura **registroCliente**
1) Membro **cidade** do membro **pessoal** da estrutura apontada por **clientePtr.**
m) Membro **estado** do membro **pessoal** da estrutura **registroCliente.** n) Membro **estado** do membro **pessoal** da estrutura apontada por **clientePtr.** o) Membro **codPostal** do membro **pessoal** da estrutura **registroCliente.** p) Membro **codPostal** do membro **pessoal** da estrutura apontada por **clientePtr.**

10.7 Modifique o programa da Fig. 10.16 para embaralhar as cartas usando um embaralhamento de alto desempe nho (como é mostrado na Fig. 10.3). Imprima o baralho resultante no formato de duas colunas mostrado na Fig. 10.4. Preceda cada carta com sua cor.

10.8 Crie a união **integer** com membros **char c, short s, int i** e **long 1.** Escreva um programa que receba valores do tipo **char, short, int** e **long** e armazene os valores em variáveis de união do tipo **union integer.** Cada variável da união deve ser impressa como **char, short, int** e **long.** Os valores sempre são impressos corretamente?

10.9 Crie a união **floatingPoint** com membros **f loat f, double** de **long double 1.** Escreva um programa que receba valores do tipo **f loat, double** e **long double** e armazene os valores em variáveis de união do tipo **union floatingPoint.** Cada variável da união deve ser impressa como **float, double** e **long double.** Os valores sempre são impressos corretamente?

10.10 Escreva um programa que desloque para a direita uma variável inteira de 4 bits. O programa deve imprimir o inteiro em bits antes e depois da operação de deslocamento. O seu sistema coloca Os ou ls nos bits liberados?

10.11 Se seu computador utiliza inteiros de 4 bits, modifique o programa da Fig. 10.7 dc forma que cie funcione com inteiros de 4 bits.

10.12 Fazer um deslocamento à esquerda de 1 bit em um inteiro **unsigned** é equivalente a multiplicar o valor por 2. Escreva a função **potencia2** que utiliza dois argumentos inteiros **numero** e **pot** e calcula

numero $* 2^{pot}$

Use o operador de deslocamento para calculai' o resultado. O programa deve imprimir os valores como inteiros e como bits.

10.13 O operador de deslocamento à esquerda pode ser usado para conter dois valores inteiros em uma variável inteira sem sinal (unsigned) de 2 bytes. Escreva um programa que receba dois caracteres do teclado e passeos para a função **possuiCaracteres.** Para colocar dois caracteres em uma variável inteira **unsigned,** atribua o primeiro caractere à variável **unsigned,** desloque a variável **unsigned** 8 posições de bits para | a esquerda e combine a variável **unsigned** com o segundo caractere usando o operador OU inclusivo bit a bit. O programa deve imprimir os caracteres em seu formato de bits antes e depois de serem colocados no inteiro **unsigned** para provar que eles foram realmente colocados nas posições corretas na variável **unsigned.**

10.14 Usando o operador de deslocamento à direita, o operador E bit a bit e uma máscara, escreva uma função **retiraCaracteres** que utilize o inteiro **unsigned** do Exercício 10.13 e decomponha-o em dois caracteres. Para retirar dois caracteres de um inteiro unsigned de dois bytes, combine o inteiro **unsigned** com a máscara **65280 (11111111 00000000)** e desloque o resultado 8 bits para a direita. Atribua o valor resultante a uma variável **char.** Depois combine o inteiro **unsigned** com a máscara **255 (0000000011111111).** Atribua o resultado à outra variável **char.** O programa deve imprimir o inteiro **unsigned** em bits antes de ser decomposto e imprimir os caracteres em bits para confirmar que eles foram obtidos corretamente.

10.15 Se seu sistema usar inteiros de 4 bytes, reescreva o programa do Exercício 10.13 para conter 4 caracteres.

10.16 Se seu sistema usar inteiros de 4 bytes, reescreva a função **retiraCaracteres** do Exercício 10.14 para obter 4 caracteres. Crie as máscaras que precisar para obter 4 caracteres deslocando o valor de 255 8 bits para a esquerda, 0, 1, 2 ou 3 vezes na variável da máscara (dependendo do byte que estiver sendo obtido).

10.17 Escreva um programa que inverta a ordem dos bits de um valor inteiro **unsigned.** O programa deve receber o valor do usuário e chamar a função **inverterBits** para imprimir os bits na ordem inversa. Imprima o valor em bits antes e depois de os bits serem invertidos para confirmar que eles foram invertidos corretamente.

10.18 Modifique a função **exibeBits** da Fig. 10.7 de forma que ela seja transportável entre sistemas usando inteiros de 2 bytes e sistemas de 4 bytes. Sugestão: Use o operador **sizeof** para determinar o tamanho de um inteiro em um determinado equipamento.

10.19 O programa a seguir usa a função **múltiplo** para determinar se o inteiro digitado no teclado é um múltiplo de outro inteiro **X.** Examine a função **múltiplo** e depois determine o valor de **X.**

```
/* Este programa determina se um valor e múltiplo de X */ #include <stdio.h> int
múltiplo(int) ;
main()
{
```

```
int y;
printf("Digite um inteiro entre 1 e 32000:   "); scanf("%d",  &y);
if  (múltiplo(y))
printf("%d e um múltiplo de X\n", y); else
printf("%d nao e um múltiplo de X\n", y) ; return 0;
}
int múltiplo(int num) {
int i, mascara = 1, mult = 1;
for  (i = 1,  i <= 10;  i + + , mascara <<= 1) if ((num & mascara)   != 0) < mult = 0; break;
}
return mult;
}
```

10.20 O que o seguinte programa faz? **#include <stdio.h>**

```
int mistério(unsigned);
main()
{
unsigned x;
printf("Digite um inteiro: "); scanf("%u",  &x);
printf("O resultado e %ã\n", mistério(x^); return 0;
}
int mistério(unsigned bits) {
unsigned i, mascara = 1 << 15,  total = 0;
for  (i = 1;   i <= 16;  i++,  bits <<= 1) if  ((bits & mascara)  == mascara) ++total;
return total % 2 == 0  ?  1  : 0;
}
```

11

Processamento de Arquivos

Objetivos

- Ser capaz de criar, ler, gravar e atualizar arquivos.
- Familiarizar-se com o processamento de arquivos de acesso seqüencial.
- Familiarizar-se com o processamento de arquivos de acesso aleatório.

Consciência... não se apresenta dividida em partes...
Uma "torrente" ou um "fluxo " são as metáforas mais naturais para descrevêla.
William James

Só posso concluir que um documento rotulado "Não Arquivar" está arquivado em um arquivo rotulado " Não Arquivar".
Senador Frank Church,
Audiência perante a Subcomissão de Inteligência do Senado Americano, 1975

Sumário

11.1 Introdução

O armazenamento de dados em variáveis e arrays é temporário; todos os dados são perdidos quando um programa termina. Os *arquivos* são usados para conservação permanente de grandes quantidades de dados. Os computadores armazenam arquivos em dispositivos secundários de armazenamento, especialmente dispositivos de armazenamento em disco. Neste capítulo, explicamos como os arquivos de dados são criados, atualizados e processados por programas em C. Examinamos aqui tanto os arquivos de acesso seqüencial como os arquivos de acesso aleatório.

11.2 A Hierarquia de Dados

Basicamente, todos os itens de dados processados por um computador são reduzidos a combinações de zeros e uns. Isso ocorre porque é simples e econômico para construir dispositivos eletrônicos que podem assumir dois estados estáveis — um dos estados representa 0 e o outro representa 1. É impressionante que as notáveis funções realizadas pelos computadores envolvam apenas as manipulações mais elementares de 0s e ls.

Os menores itens de dados em um computador podem assumir o valor 0 ou o valor 1. Tal iten de dados é chamado um *bit* (abreviação de Binary digít, ou dígito binário — um dígito que pode **assadl** um de dois valores). Os circuitos computacionais realizam várias manipulações simples de bits como determinar o valor de um bit, redefinir o valor **de** um bit e inverter o valor de um bit (de 1 para 0 ou de 0 para 1).

Torna-se complicado para os programadores trabalhar com dados na forma de baixo nível como são os bits. Em vez disso, os programadores preferem trabalhar com dados na forma de *dígitos ch* ■. (i.e., 0, 1,2, 3, 4, 5, 6, 7, 8 e 9), *letras* (i.e., de A até Z e de a até z) e *símbolos especiais* (i.e., $, @, %, &,*,(,),—,+,",:, ?, /e muitos outros). Dígitos, letras e símbolos especiais são chamados *caracteres.* O conjunto de todos os caracteres que podem ser usados para escrever programas e representar itens de dados em um determinado computador é chamado *conjunto de caracteres* daquele computador. Como os computadores podem processar apenas ls e Os, qualquer caractere do conjunto de caracteres de um computador é representado por um padrão de ls e Os (chamado um *byte).* Atualmente, os bytes são compostos normalmente de oito bits. Os programadores criam programas e itens de dados como caracteres; os computadores manipulam e processam esses caracteres como padrões de bits.

Da mesma forma que os caracteres são compostos de bits, os *campos* são compostos de caracteres. Um campo é um grupo de caracteres que possui um significado. Por exemplo, um campo consistindo unicamente em letras maiúsculas e minúsculas pode ser usado para representar o nome de uma pessoa.

Os itens de dados processados pelos computadores formam uma *hierarquia de dados* na qual os itens de dados se tornam maiores e mais complexos na estrutura à medida que evoluímos de bits para caracteres, campos e assim por diante.

Um *registro* (i.e., uma **struct** em C) é composto de vários campos. Em um sistema de folha de pagamento, por exemplo, um registro de um determinado empregado pode consistir nos seguintes campos:

1. Número de Previdência Social
2. Nome
3. Endereço
4. Valor do salário por hora
5. Número de dispensas solicitadas
6. Total de vencimentos no presente ano
7. Total de impostos federais retido na fonte etc.

Dessa forma, um registro é um grupo de campos relacionados entre si. No exemplo anterior, cada um dos campos pertence ao mesmo empregado. Obviamente, uma companhia em particular pode ter muitos empregados e terá um registro da folha de pagamento para cada empregado. Um *arquivo* é um grupo de registros relacionados. Um arquivo de folha de pagamento de uma companhia contém um registro para cada empregado. Assim, o arquivo de folha de pagamento de uma pequena empresa deve conter apenas 22 registros, ao passo que o arquivo da folha de pagamento de uma grande empresa pode conter 100.000 registros. Não é raro uma organização ter centenas ou mesmo milhares de arquivos e cada um deles possuir milhões ou até bilhões de caracteres de informações. Com a crescente popularidade de discos a laser e da tecnologia multimídia, em breve serão comuns arquivos que chegam aos trilhões de bytes. A Fig. 11.1 ilustra a hierarquia de dados

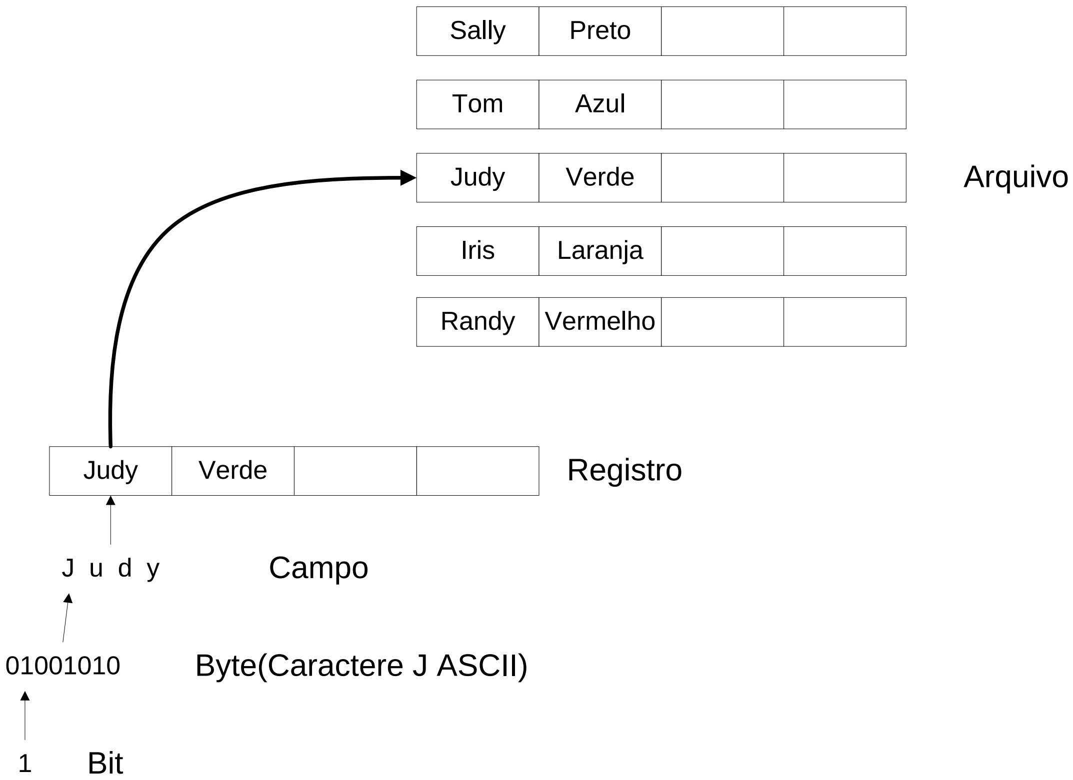


Fig. 11.1 A hierarquia de dados.

Para facilitar a recuperação de registros específicos de um arquivo, pelo menos um campo em cada registro é escolhido como *chave* (ou *campo-chave) dos registros. A* chave dos registros identifica um registro como pertencendo a uma determinada pessoa ou entidade. Por exemplo, no registro de folha de pagamento descrito nesta seção, o número de inscrição na Previdência Social seria escolhido normal mente como chave dos registros.

Há muitas maneiras de organizar registros em um arquivo. O tipo mais popular de organização é chamado *arquivo seqüencial* no qual normalmente os registros são armazenados em ordem segundo o campo-chave dos registros. Em um arquivo de folha de pagamentos, geralmente os registros são colocados em ordem segundo o número de inscrição na Previdência Social. O registro do primeiro empregado no arquivo contém o menor número de Previdência Social e os registros subseqüentes possuem números de Previdência Social em ordem crescente.

A maioria das empresas utiliza muitos arquivos diferentes para armazenar dados. Por exemplo, as companhias podem ter arquivos de folhas de pagamento, arquivos de contas a receber (listando o dinheiro devido pelos clientes), arquivos de contas a pagar (listando o dinheiro devido aos fornecedores), arquivos de inventário (listando características a respeito de todos os itens gerenciados pela empresa muitos outros tipos de arquivos. Algumas vezes, um grupo de arquivos relacionados entre si é chamado, *banco de dados.* Um conjunto de programas que se destina a criar c gerenciar bancos de dados é chamado *sistema de gerenciamento de banco de dados* (SGBD, ou *database management system, DBMS).*

11.3 Arquivos e Fluxos

A linguagem C visualiza cada arquivo simplesmente como um fluxo seqüencial de bytes (Fig. 11.2). Cada arquivo termina ou com um *marcador de final de arquivo (end-of-file marker),* ou em usa específico gravado em uma estrutura administrativa de dados, mantida pelo sistema. Quando um arquivo é *aberto,* um fluxo é associado àquele arquivo. Três arquivos e seus respectivos fluxos são abertos automaticamente quando a execução de um programa se inicia — o de *entrada padrão,* o de saída *padrão* e o de *erro* (ou *exibição de erros) padrão.* Os fluxos fornecem canais de comunicação entre arquivos e programas. Por exemplo, o fluxo padrão de entrada permite que um programa leia dados do teclado e o fluxo padrão de saída permite que um programa imprima dados na tela.

Abrir um arquivo retorna um ponteiro para uma estrutura **FILE** (definida em < **stdio.h>) que contém** informações usadas para processar o arquivo. Esta estrutura inclui um *descritor do arquivo,* i.e.. **un:** em um array do sistema operacional chamado *tabela de arquivos abertos.* Cada elemento do array comtém um *bloco de controle de arquivo (BCA,* ou *file control block, FCB)* que o sistema operacional usa para administrar um arquivo em particular. O padrão de entrada, o padrão de saída e o padrão de erros são manipulados por meio dos ponteiros de arquivos **stdin, stdout** e **stderr.**

A biblioteca padrão fornece muitas funções para ler dados de arquivos e também para gravar dados em arquivos. A função **fgetc,** como **getchar,** lê um caractere de um arquivo. A função **fgetc re**cebe como argumento um ponteiro **FILE** do arquivo para o qual o caractere será lido. A chamada do **fgetc (stdin)** lê um caractere **de stdin** — o dispositivo padrão de entrada. Essa chamada é equivalente à chamada a **getchar ()**. A função **fputc,** como **putchar,** escreve um caractere em um arquivo. A função **fputc** recebe como argumentos um caractere a ser escrito e um ponteiro para o arquivo ao qual o caractere será escrito. A chamada à função **f pute (' a ' , stdout)** escreve o caractere '**a'** em **stdout** — o dispositivo padrão de saída. Essa chamada é equivalente **aputchar ('a').**

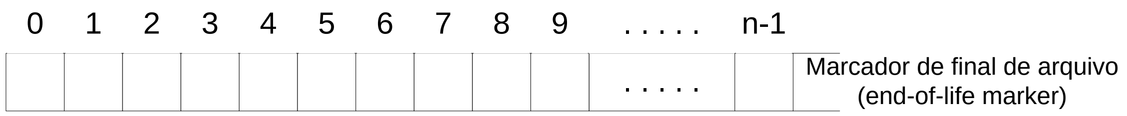


Fig. 11.2 Como a linguagem C visualiza um arquivo de n bytes,

Várias outras funções usadas para ler dados do dispositivo padrão de entrada e gravar (escrever) dados no dispositivo padrão de saída possuem funções com nomes similares para efetuarem o processamento de arquivos. As funções **f gets** e **f puts,** por exemplo, podem ser usadas para ler uma linha de um arquivo e gravar uma linha em um arquivo, respectivamente. As funções equivalentes para ler da entrada padrão e gravar na saída padrão, **gets** e **puts,** foram analisadas no Capítulo 8. Nas seções que se seguem, apresentamos as funções equivalentes a **scanf** e **printf** para o processamento de arquivos — **fscanf** e **fprintf**. Posteriormente ainda neste capítulo analisamos as funções **fread** e **fwrite.**

11.4 Criando um Arquivo de Acesso Seqüencial

A linguagem C não impõe estrutura a um arquivo. Assim, noções como um registro de um arquivo não existem como parte da linguagem C. Portanto, o programador deve fornecer qualquer estrutura de arquivo para satisfazer as exigências de uma aplicação específica. No exemplo a seguir, vemos como o programador pode impor uma estrutura de registros em um arquivo.

O programa da Fig. 11.3 cria um arquivo simples de acesso seqüencial que pode ser usado em um sistema de recebimento de contas para ajudar a controlar as quantias devidas pelos clientes devedores de uma companhia. Para cada cliente, o programa obtém um número de conta, o nome do cliente e o saldo do cliente (i.e., a quantia que o cliente deve à companhia por bens e serviços recebidos no passado). Os dados obtidos para cada cliente constituem um "registro" para aquele cliente. O número da conta é usado como campo-chave dos registros nessa aplicação — o arquivo será criado e mantido segundo a ordem dos números de contas. Esse programa assume que o usuário fornece os registros na orderm dos números das contas. Em um sistema grande de contas a receber, seria fornecido um recurso de ordenação para que o usuário pudesse entrar com os registros em qualquer ordem. Os registros seriam então ordenados e gravados no arquivo.

```
/*Cria um arquivo seqüencial */
#include <stdio.h>

main() {

        int conta;
        char nome[30];
        float saldo;
        FILE *cfPtr;      /* cfPtr = ponteiro do arquivo clientes.dat */

        if ((cfPtr = fopen("clientes.dat",   "w"))  == NULL)
                printf("Arquivo nao pode ser aberto\n");
        else {
                printf("Digite a conta, o nome e o saldo.\n");
                printf("Digite EOF para encerrar a entrada de dados.\n”);
                printf("? ");
                scanf("%d%s%f",  &conta, nome, &saldo);

                while  (!feof(stdin)) {
                        fprintf(cfPtr,   "%d %s %.2f\n",conta, nome,  saldo);

                        printf("? ");
                        scanf("%d%s%f", fcconta, nome, &saldo);
                }

                fclose(cfPtr);
        }
        return 0;
}
```

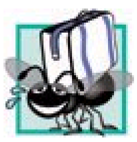
```
Digite a conta, o nome e o saldo.
Digite EOF para encerrar a entrada de dados.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?
```

Fig. 11.3 Criando um arquivo seqüencial,

Agora, vamos examinar esse programa. A instrução

FILE *cfPtr;

afirma que **cfPtr** é um ponteiro para uma estrutura **FILE.** O programa em C administra cada arquivo com uma estrutura **FILE** separada. O programador não precisa saber os detalhes da estrutura **FILE para** usar os arquivos. Em breve veremos precisamente como a estrutura **FILE** conduz indiretamente para o bloco de controle de arquivo (BCA) do sistema operacional de um arquivo.



Dicas de portabilidade 11.1

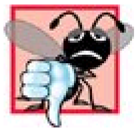
*A estrutura **FILE** varia conforme o sistema operacional (i.e., os membros da estrutura variam entre sistemas conforme o modo como cada sistema manipula seus arquivos).*

Cada arquivo aberto deve ter um ponteiro do tipo **FILE** declarado separadamente e que é usado fazer referência ao arquivo. A linha

if ((cfPtr = fopen("clientes.dat", "w")) == NULL)

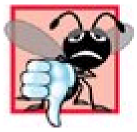
indica o arquivo — **"clientes.dat"** — a ser usado pelo programa e estabelece um "canal de comunicação" com o arquivo. Ao ponteiro de arquivo **cf Ptr** é atribuído um ponteiro para a estrutura FILE do arquivo aberto com **fopen.** A função **fopen** exige dois argumentos: um nome de arquivo e um *modo de abertura de arquivo.* O modo de abertura de arquivo " w" indica que o arquivo deve ser aberto para *gravação (writing).* Se um arquivo não existir e for aberto para gravação, **f open** cria o arquivo. Se um arquivo existente for aberto para gravação, o conteúdo do arquivo é eliminado sem qualquer aviso. No programa, a estrutura **if** é usada para determinar se o ponteiro de arquivo **cf Ptr** é **NULL (i.e., o** arquivo não está aberto). Se ele for **NULL,** é impressa uma mensagem de erro e o programa é encerrado.

Caso contrário, a entrada é processada e gravada no arquivo.



Erro comun de programação 11.1

Abrir para gravação ("w") um arquivo existente quando, na realidade, o usuário desejava preservar o arquivo: o conteúdo do arquivo é eliminado sem qualquer aviso.



Erro comun de programação 11.2

Esquecer-se de abrir um arquivo antes de tentar fazer referência a ebe em um programa.

O programa pede para o usuário digitar os vários campos para cada registro ou para indicar o fim do arquivo (EOF) quando a entrada de dados for concluída. A Fig. 11.4 lista as combinações de teclas para digitar o fim do arquivo em vários sistemas computacionais.

A linha

while (!feof(stdin))

usa a função **feof** para determinar se o *indicador de fim de arquivo* foi digitado para o arquivo ao qual **stdin** se refere. O indicador de fim de arquivo informa ao programa que não há mais dados a serem processados. No programa da Fig. 11.3, o indicador de fim de arquivo é definido para o dispositivos de entrada padrão quando o usuário digita a combinação de teclas para fim de arquivo. O argumento para a função **feof** é um ponteiro para o arquivo em que o indicador de fim de arquivo **(stdin** nesse caso) será testado. A função retorna um valor diferente de zero (verdadeiro) depois de o indicador de fim de arquivo ser definido; caso contrário, é retornado o valor zero. A estrutura **while** que inclui a chamada **feof** nesse programa continua a ser executada enquanto o indicador de fim de arquivo não for estabelecido.

Sistema computacional	***Combinação de teclas***
Sistemas UNIX	*<return><ctrl>d*
IBM PC e compatíveis	*<ctrl>z*
Macintosh	*<ctrl>d*
VAX (VMS)	*<ctrl>z*

Fig 11.4 Combinações de teclas que indicam fim de arquivo (EOF, end-of-file) para os vários sistemas computacionais populares.

A instrução

fprintf(cfPtr, *"%d %s* %.2f\n", conta, nome, saldo);

grava dados no arquivo **clientes. dat.** Os dados podem ser recuperados mais tarde por um programa desenvolvido para ler o arquivo (veja a Seção 11.5). A função **fprintf** é equivalente a **printf exceto** que **fprintf** também recebe como argumento um ponteiro para o arquivo no qual os dados serão gravados.

Erro comun de programação 11.3

Usar o ponteiro errado de arquivo para fazer referência a um arquivo.

Boa prática de programação 11.1

Certifique-se de que as chamadas a funções de processamento de arquivo em um programa contêm os ponteiros corretos de arquivos.

Depois de o usuário fornecer o indicador de fim de arquivo, o programa fecha o arquivo **clientes.dat** com **fclose** e é encerrado. A função **fclose** também recebe o ponteiro de arquivo (em vez do nome do arquivo) como argumento. Se a função f**close** não for chamada explicitamente, é normal que o sistema operacional feche o arquivo quando o programa for encerrado. Isso é um exemplo de "housekeeping" ("faxina") do sistema operacional.

Boa prática de programação 11.2

Feche explicitamente os arquivos tão logo saiba que o programa não fará nova referência a eles.

Dica de desempenho 11.1

Fechar um arquivo pode liberar recursos pelos quais outros usuários podem estar esperando.

No exemplo de execução do programa da Fig. 11.3, o usuário digita informações para cinco contas e depois fornece o indicador de final de arquivo para informar que a entrada de dados foi concluída. O exemplo de execução não mostra como os registros de dados aparecem realmente no arquivo. Para verificar se o arquivo foi criado satisfatoriamente, na próxima seção apresentamos um programa que lê o arquivo e imprime seu conteúdo.

A Fig. 11.5 ilustra o relacionamento entre os ponteiros **FILE,** as estruturas **FILE** e os BCAs na memória. Quando o arquivo **"clientes.dat**" é aberto, um BCA para o arquivo é copiado na memória.

A figura mostra a conexão entre o ponteiro de arquivo retornado por **f open** e o BCA usado pelo sistema operacional para administrar o arquivo.

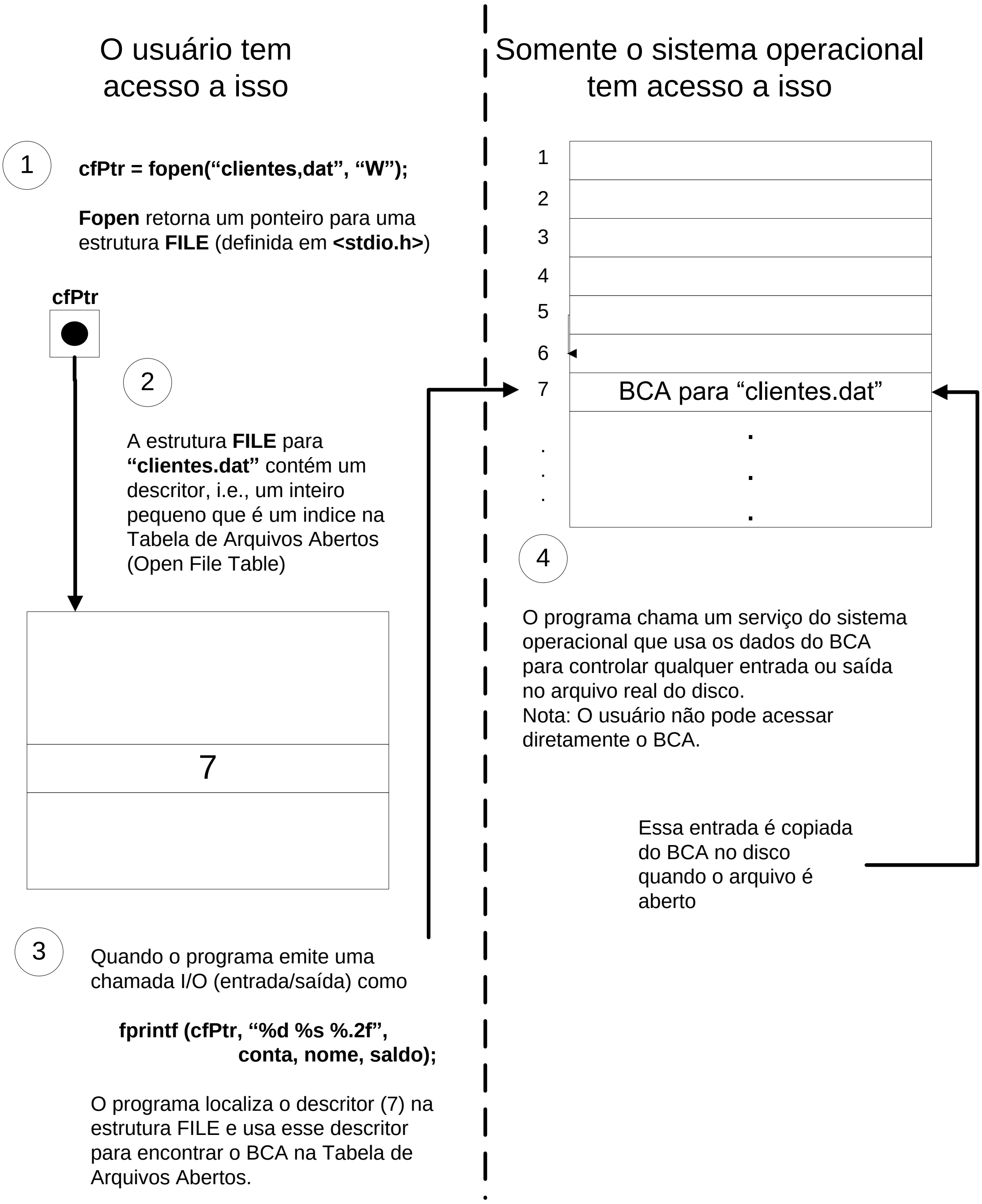
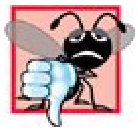


Fig. 11.5 O relacionamento entre os ponteiros **file**, as estruturas **file** e os BCAs,

Os programas podem não processar nenhum arquivo, processar um arquivo ou processar vários arquivos. Cada arquivo usado em um programa deve ter um nome exclusivo e terá um ponteiro diferente de arquivo retornado por **f open.** Todas as funções subseqüentes de processamento de arquivo depois de o arquivo ser aberto devem se referir ao arquivo com o ponteiro de arquivo apropriado. Os arquivos podem ser abertos de várias maneiras. Para criar um arquivo ou eliminar o conteúdo de um arquivos antes da gravação dos dados, abra o arquivo para gravação (" w"). Para ler um arquivo existente, abre-o para leitura **("r").** Para adicionar registros ao final de um arquivo existente, abra o arquive anexação (**"a"**). Para abrir um arquivo de forma que ele possa ser gravado e lido, abra o arquivo com um dos três modos de atualização — **"r+", "w+"** ou "**a+**". O modo "**r+**" abre um arquivo para leitura e gravação. O modo

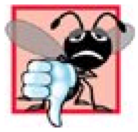
"w+" cria um arquivo para leitura e gravação. Se o arquivo já existir, o arquivo é aberto e o conteúdo atual é eliminado. O modo " **a** +" abre um arquivo para leitura e gravação — toda gravação é feita no final do arquivo. Se o arquivo não existir, é criado.

Se ocorrer um erro durante a abertura de um arquivo de qualquer modo, **f open** retorna **NULL.** Alguns erros possíveis são:



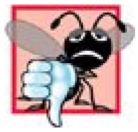
Erro comun de programação 11.4

Abrir para leitura um arquivo não-existente.



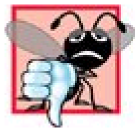
Erro comun de programação 11.5

Abrir para leitura ou gravação um arquivo sem ter garantido os direitos apropriados de acesso em relação ao arquivo.



Erro comun de programação 11.6

Abrir para gravação um arquivo quando não houver espaço disponível em disco. A Fig. 11.6 lista os modos de abrir arquivos.



Erro comun de programação 11.7

Abrir um arquivo com o modo incorreto pode levar a erros terríveis. Por exemplo, abrir um arquivo no modo de gravação (" w") quando ele deve ser aberto no modo de atualização (" r+") faz com que o conteúdo do arquivo seja eliminado.



Boa prática de programação 11.3

Abrir um arquivo apenas para leitura (e não atualização) se seu conteúdo não deve ser modificado. Isso evita modificações não-intencionais do conteúdo do arquivo. Isso é outro exemplo do princípio do privilégio mínimo.

11.5 Lendo Dados de um Arquivo de Acesso Seqüencial

Os dados são armazenados em arquivos de modo que possam ser recuperados para processamento quando necessário. A seção anterior demonstrou como criar um arquivo para acesso seqüencial. Nesta seção, analisaremos como ler dados seqüencialmente de um arquivo.

Modo	Descrição
r	Abre um arquivo para leitura
w	Cria um arquivo para gravação. Se o arquivo já existir, elimina o conteúdo atual.
a	Anexa: abre ou cria um arquivo para gravação no final do arquivo
r+	Abre um arquivo para atualização (leitura e gravação)
w+	Cria um arquivo para atualização. Se o arquivo já existir, elimina o conteúdo atual.
a+	Anexa: abre ou cria um arquivo para atualização; a gravação é feita no final do arquivo

Fig. 11.6 Modos de abertura de arquivos,

```
/* Lendo e imprimindo um arquivo seqüencial */
#include <stdio.h>

main() {

	int conta;
	char nome[30];
	float saldo;
	FILE *cfPtr;	/* cfPtr = ponteiro do arquivo clientes.dat */

	if  ((cfPtr = fopen("clientes.dat",   "r"))  == NULL)
		printf("Arquivo nao pode ser aberto\n");

	else {
		printf("%-10s%-13s%s\n",   "Conta",   "Nome", "Saldo");
		fscanf(cfPtr,   "%d%s%f",  &conta, nome,  &saldo);

		while  (!feof(cfPtr)) {
			printf("%-10d%-13s%7.2f\n",  conta, nome, saldo);
			fscanf(cfPtr,   "%d%s%f",  &conta, nome,  &saldo);
		}
		fclose(cfPtr);
	}
	return 0;
}
```

```
Conta     Nome         Saldo
100       Jones        24.98
200       Doe          345.67
300       White        0.00
400       Stone        -42.16
500       Rich         224.62
```

Fig. 11.7 Lendo e imprimindo um arquivo seqüencial

O programa da Fig. 11.7 lê registros do arquivo " **clientes . dat**" criado pelo programa da Fig. 11.3 e imprime o conteúdo nos registros. A instrução

FILE *cfPtr;

indica que **cf Ptr** é um ponteiro para um **FILE.** A linha

if ((cfPtr = fopenCclientes.dat", "r")) == NULL)

tenta abrir o arquivo " **clientes.dat**" para leitura **("r")** e determina se o arquivo foi aberto corretamente (i.e., **f open** não retorna **NULL).** A instrução

fscanf(cfPtr, *"%â%s%£",* **&conta, nome, &saldo);**

lê um "registro" do arquivo. Uma função **f scanf** é equivalente à função **scanf** exceto que **fscanf** recebe um argumento ponteiro para o arquivo em que os dados são lidos. Depois de a instrução anterior ser executada pela primeira vez, **conta** terá o valor **100, nome** terá o valor " **Jones**" e saldo terá o valor **24.98.** Cada vez que a segunda instrução **fscanf** é executada, outro registro é lido do arquivo, e **conta, nome** e **saldo** assumem novos valores. Quando o final do arquivo é encontrado, o arquivo é fechado e o programa é encerrado.

Para recuperar dados seqüencialmente de um arquivo, normalmente um programa começa a leitura no início do arquivo e lê todos os dados consecutivamente até chegar ao dado procurado. Pode ser desejável processar os dados seqüencialmente várias vezes (desde o início do arquivo) durante a execução de um programa. Uma instrução como

rewind(cfPtr) ;

faz com que o *ponteiro de posição do arquivo* do programa — indicando o número do próximo byte do arquivo a ser lido ou gravado — seja reposicionado no início do arquivo (i.e., byte 0) apontado por **cf**Pt r. O ponteiro de posição do arquivo não é realmente um ponteiro. Em vez disso, ele é um valor inteiro que especifica a posição do byte no qual ocorrerá a próxima leitura ou gravação. Algumas vezes ele é chamado *offset de arquivo.* O ponteiro de posição de arquivo é um membro da estrutura **FILE** associado a cada arquivo.

Agora apresentamos um programa (Fig. 11.8) que permite que um gerente de crédito obtenha listas de clientes com crédito zero (i.e., clientes que não devem dinheiro algum), clientes com saldos credores (i.e., clientes aos quais a companhia deve dinheiro) e clientes com saldos devedores (i.e., clientes que devem dinheiro à companhia por bens e serviços recebidos). Um saldo credor é uma quantia negativa um saldo devedor é uma quantia positiva.

```
/* Programa de consulta de credito */
#include <stdio.h>

main()
{

        int pedido, conta;
        float saldo;
        char nome[30];
        FILE *cfPtr;

        if ( (cfPtr = fopenCclientes.dat",   "r")) == NULL)
                printf("Arquivo nao pode ser aberto\n");
        else {
                printf("Digite pedido\n"
                        "1 — Lista contas com saldo zero\n"
                        "2 — Lista contas com saldo credor\n"
                        "3 — Lista contas com saldo devedor\n"
                        "4 — Encerra o programa\n? ");
                scanf("%d",  fcpedido);

                while (pedido != 4) {
                fscanf(cfPtr,  "%d%s%f", &conta, nome, &saldo);
                switch (pedido) {

                        case 1:
                                printf("\nContas com saldo zero:\n");
                                while (!feof(cfPtr)) {
                                if  (saldo == 0)
                                        printf ("%-10d%-13s%7.2f \n", conta, nome, saldo);
                                fscanf(cfPtr,   "%d%s%f",&conta, nome, &saldo);
                                break;

                        case 2:
                                printf("\nContas com saldo credor:\n");
                                while (!feof(cfPtr)){
                                        if (saldo < 0)
                                                printf("%-10d%-13s%7.2f\n", conta,  nome,  saldo);
                                        fscanf(cfPtr,   "%d%s%f", &conta, nome, &saldo);
                                }
                                break;

                        case 3:
                                printf("\nContas com saldo devedor:\n");
                                while (!feof(cfPtr)) {
                                        if (saldo > 0)
                                                printf("%-10d%-13s%7.2f\n", conta, nome,  saldo);
                                        fscanf(cfPtr,   "%d%s%f", &conta, nome,  &saldo);
                                }
                                break;
                }
                rewind(cfPtr);
                printf("\n? ");
                scanf("%d",  &pedido);
        }
```

```
            printf("Fim do programa.\n");
            fclose(cfPtr);
            }
            return 0;
}
```

Fig. 11.8 Programa de consulta de crédito .

O programa exibe um menu e permite que o gerente de crédito entre com uma das três opções para obter as informações dos clientes. A Opção 1 produz uma lista de contas com saldo zero. A Opção 2 produz uma lista das contas com saldos credores. A Opção 3 produz uma lista de contas com saldos devedores. A Opção 4 termina a execução do programa. Um exemplo de saída é mostrado na Fig. 11.9.

Observe que os dados nesse tipo de arquivo seqüencial não podem ser modificados sem o risco de destruir outros dados no arquivo. Por exemplo, se o nome **"White"** precisasse ser modificad **"Worthington",** o nome antigo simplesmente não poderia ser sobrescrito. **O** registro para White foi gravado no arquivo como

300 White 0.00

Se o registro precisasse ser reescrito com um novo nome, iniciando na mesma posição do arquivo, o registro seria

300 Worthington 0.00

O novo registro é maior do que o registro original. **Os** caracteres além do segundo **"o"** em **"WorthingGton"** sobrescreveriam o início do próximo registro seqüencial no arquivo. **O** problema aqui é que no modelo formatado de entrada/saída usando **fprintf** e **f scanf**, o tamanho dos campos — e portanto dos registros — pode variar. Por exemplo, 7, 14, — 117, 2074 e 27383 são todos **ints** armazenados internamente no mesmo número de bytes, mas são impressos na tela ou no disco, por meio de **fprintf,** como campos de tamanhos diferentes.

Assim sendo, o acesso seqüencial com **fprintf** e **f scanf** não é usado normalmente para atualizar registros no local onde eles se encontram. Em vez disso, normalmente o arquivo inteiro é reescrito.

Para fazer a alteração de nome mencionada anteriormente, os registros antes de **3 0 0 Whi te 0.00 em** tal arquivo seqüencial seriam copiados para um novo arquivo, o novo registro seria gravado e os registros após **300 White 0.00** seriam copiados para o novo arquivo. Isso exige o processamento de todos os registros do arquivo para atualizar um registro.

```
Digite pedido
1 - Lista contas com saldo zero
2 - Lista contas com saldo credor
3 - Lista contas com saldo devedor
4 - Encerra o programa ? 1
Contas com saldo zero: 300 White 0.00
? 2
Contas com saldo credor: 400 Stone -42.16
? 3
Contas com saldo devedor: 100 Jones 24.98
200 Doe 345.67
500 Rich 224.62
? 4
Fim do programa.
```

Fig. 11.9 Exemplo de saída do programa de consulta de crédito,

11.6 Arquivos de Acesso Aleatório

Como mencionamos anteriormente, os registros em um arquivo criado com a função de saída formatado **fprintf** não possuem necessariamente o mesmo tamanho. Entretanto, normalmente os registros de um *arquivo de acesso aleatório* possuem o mesmo tamanho e podem ser acessados diretamente (e, dessa forma, rapidamente) sem passar por outros registros. Isso faz com que os arquivos de acesso aleatórios sejam apropriados para sistemas de reservas de vôos, sistemas de bancos, sistemas de ponto de venda e outros tipos de *sistemas de processamento de transações* que exigem acesso rápido a dados específicos. Há outras maneiras de implementar arquivos de acesso aleatório, mas limitaremos nossa análise ao método simples que usa registros de comprimento fixo.

Como todos os registros de um arquivo de acesso aleatório normalmente possuem o mesmo comprimento, a localização exata de um registro relativo ao início do arquivo pode ser calculada como uma função da chave dos registros. Veremos em breve como isso facilita o acesso imediato a registros específicos, mesmo em arquivos grandes.

A Fig. 11.10 ilustra uma maneira de implementar um arquivo de acesso aleatório. Tal arquivo é como um trem de carga com muitos vagões — alguns vazios e outros carregados. Cada vagão no trem possue o mesmo comprimento.

Os dados podem ser inseridos em um arquivo de acesso aleatório sem que sejam destruídos outros dados no arquivo. Os dados armazenados previamente podem ser atualizados ou excluídos sem reescrver todo o arquivo. Nas seções que se seguem explicamos como criar um arquivo de acesso aleatório e entrar com os dados, ler os dados tanto seqüencial quanto aleatoriamente, atualizar os dados e excluir o dados que não mais se fizerem necessários.

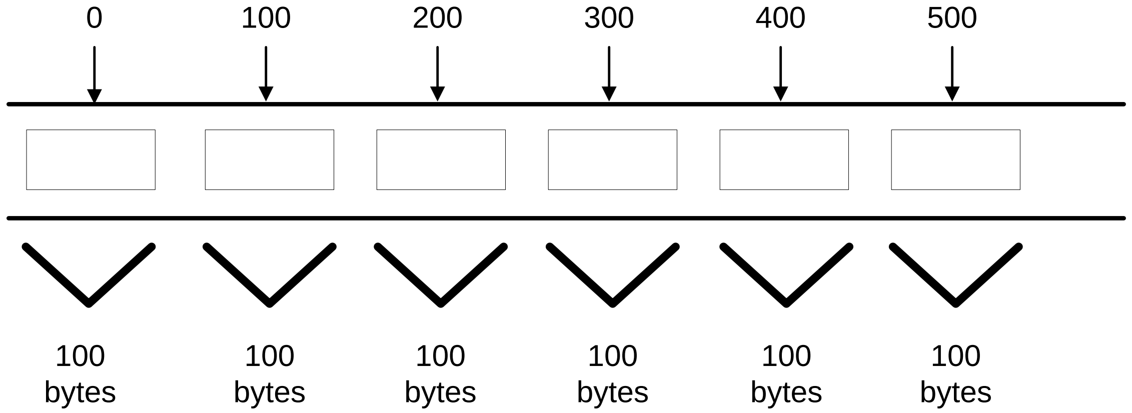


Fig. 11.10 Vista de um arquivo de acesso aleatório com registros de comprimento fixo.

11.7 Criando um Arquivo de Acesso Aleatório

A função **fwrite** transfere para um arquivo um número especificado de bytes, iniciando em um determinado local da memória. Os dados são gravados iniciando no local do arquivo indicado pelo ponteiro de posição do arquivo. A função **f read** transfere um número determinado de bytes de um local específico da memória, definido pelo ponteiro de posição do arquivo, para uma área da memória, iniciando em um endereço indicado. Agora, ao gravar um inteiro, em vez de usar

fprintf(fPtr, *"%d"*, **numero);**

que poderia imprimir o mínimo de 1 dígito ou o máximo de 11 dígitos (10 dígitos mais um sinal, exigindo, cada um deles, um byte para armazenamento) para um inteiro de 4 bytes, podemos usar

fwrite(Snumero, sizeof(int), 1, **fPtr);**

que sempre grava 4 bytes (ou 2 bytes em um sistema com inteiros de 2 bytes) da variável **numero no** arquivo representado por f**Ptr** (em breve explicaremos o argumento 1). Posteriormente, **fread** pode ser usado para ler 4 daqueles bytes em uma variável inteira **numero.** Embora **f read** e **fwrite leiam** e gravem dados como inteiros em um formato de tamanho fixo, em vez de tamanho variável, os dados que eles manipulam são processados no formato de "dados computacionais brutos" (i.e., bytes de dados) e não no formato de leitura humana proporcionado por **printf** e **scanf.**

As funções **fread** e **fwrite** são capazes de ler e gravar arrays de dados de e para um disco. O terceiro argumento, tanto de **fread** quanto de **fwrite,** é o número de elementos do array que deve ser lido do disco ou gravado nele. A chamada anterior à função **fwrite** grava um inteiro simples no disco, portanto o terceiro argumento é 1 (como se um elemento do array estivesse sendo gravado).

Os programas de processamento de arquivos raramente gravam um campo isolado em um arquivo. Normalmente eles gravam uma **struct** de cada vez, como mostraremos nos exemplos a seguir.

Considere o seguinte enunciado de um problema:

Crie um sistema de processamento de créditos capaz de armazenar até 100 registros de comprimento fixo. Cada registro deve consistir em um número de conta que será usado como chave dos registros, um sobrenome, um primeiro nome e um saldo. O programa resultante deve ser capaz de atualizar uma conta, inserir um novo registro de conta, excluir uma conta e listar todos os registros de contas em um arquivo formatado de texto para impressão. Use um arquivo de acesso aleatório.

As várias seções que se seguem apresentam as técnicas necessárias para criar o programa de processamento de créditos. O programa da Fig. 11.11 mostra como abrir um arquivo de acesso aleatório, definir um formato de registro usando **struct,** gravar dados em um disco e fechar o arquivo. Esse programa inicializa 100 registros do arquivo " **credito. dat**" com **structs** vazias, usando a função **fwrit**e. Cada **struct** vazia contém **0** para número da conta, **NULL** (representado por aspas duplas vazias) para o

sobrenome, **NULL** para o primeiro nome e **0 . 0** para o saldo. O arquivo é inicializado dessa maneira para criar espaço no disco no qual o arquivo será armazenado e para tornar possível determinar se um registro contém dados.

```
/* Criando seqüencialmente um programa de acesso aleatório */
#include <stdio.h>

struct dadosCliente {
          int numConta;
          char sobrenome[15];
          char primNome[10];
          float saldo;
};

main() {

          int i;
          struct dadosCliente clienteNulo = {0,   "",   ""; 0.0};
          FILE *cfPtr;

          if  ((cfPtr = fopen("credito.dat",   "w"))  == NULL)
                    printf("Arquivo nao pode ser aberto.\n");
          else {

                    for (i = 1;  i <= 100; i++)
                              fwrite(fcclienteNulo, sizeof(struct dadosCliente), 1, cfPtr);

                    fclose  (cfPtr);
          }

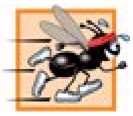
          return 0;
}
```

Fig. 11.11 Criando seqüencialmente um programa de acesso aleatório.

A função **fwrite** grava um bloco (número específico de bytes) de dados em um arquivo. Em nosso programa, a instrução

fwrite(&clienteNulo, sizeof(struct dadosCliente), 1, **cfPtr);**

faz com que a estrutura **clienteNulo** de tamanho **sizeof (struct dadosCliente)** seja gravada no arquivo apontado por **cf Ptr.** O operador de **sizeof** retorna o tamanho em bytes do objeto contido entre parênteses (nesse caso **struct dadosCliente).** O operador **sizeof** é um operador unário de tempo de compilação que retorna um inteiro sem sinal. O operador **sizeof** pode ser usado para determinar o tamanho em bytes de qualquer tipo de dado ou expressão. Por exemplo, **sizeof (int)** é usado para determinar se um inteiro está armazenado em 2 ou 4 bytes de um determinado computador.



Dica de desempenho 11.2

*Muitos programadores pensam erradamente que **sizeof** é uma função e que usá-lo gera o overhead de tempo de execução da chamada de uma função. Não existe tal overhead porque **sizeof** é um operador de tempo de compilação.*

A função **fwrite** pode até ser usada para gravar vários elementos de um array de objetos. Para gravar vários elementos de um array, o programador deve fornecer, como primeiro argumento da chamada a **fwrite,** um ponteiro para um array e especificar como terceiro argumento daquela chamada o número de elementos a serem gravados. No exemplo anterior, **fwrite** foi usado para gravar um único objeto que não era elemento de um array. Gravar um objeto isolado é equivalente a gravar um elemento de um array, daí a presença do 1 na chamada de **fwrite.**

11.8 Gravando Dados Aleatoriamente em um Arquivo de Acesso Aleatório

O programa da Fig. 11.12 grava dados no arquivo " **credito. dat**". Ele usa a combinação de **fseek** com **fwrite** para armazenar dados em locais específicos do arquivo. A função **fseek** define o ponteiro de posição do arquivo em um local específico do arquivo e depois **fwrite** grava os dados. Um exemplo de execução é mostrado na Fig. 11.13.

```
/* Escrevendo em um arquivo de acesso aleatório */
#include <stdio.h>

struct dadosCliente {
        int numConta;
        char sobrenome[15];
        char primNome[10];
        float saldo;
};

main () {

        FILE *cfPtr;
        struct dadosCliente cliente;

        if  ((cfPtr = fopen("credito.dat",   "r+"))  == NULL)
                printf("Arquivo nao pode ser aberto.\n");
        else {
                printf("Digite o numero da conta"
                        "   (1 a 100,   0 para encerrar dados)\n? ");
                scanf("dV, &cliente.numConta);

                while  (cliente.numConta != 0) {
                      printf("Digite sobrenome, primeiro nome e saldo\n? ");
                      scanf("%s%s%f",  &cliente.sobrenome,&cliente.primNome, &cliente.saldo);
                      fseek(cfPtr,   (cliente.numConta - 1)*sizeof(struct dadosCliente), SEEK_SET);
                      fwrite(&cliente,sizeof(struct dadosCliente),  1,  cfPtr);
                      printf("Digite o numero da conta\n? ");
                      scanf("%d", &cliente.numConta);
                }
        }
        fclose(cfPtr);
        return 0;
}
```

Fig. 11.12 Escrevendo dados aleatoriamente em um arquivo de acesso aleatório.

```
Digite o numero da conta (1 a 100, 0 para encerrar dados) ? 37
Digite sobrenome, primeiro nome e saldo ?
Barker Doug 0.00

Digite o numero da conta ? 29
Digite sobrenome, primeiro nome e saldo ?
Brown Nancy -24.54

Digite o numero da conta ? 96
Digite sobrenome, primeiro nome e saldo ?
Stone Sam 34.98

Digite o numero da conta ? 88
Digite sobrenome, primeiro nome e saldo ?
Smith Dave 258.34

Digite o numero da conta ? 33
Digite sobrenome, primeiro nome e saldo ?
Dunn Stacey 314.33

Digite o numero da conta ? 0
```

Fig. 11.13 Exemplo de execução do programa da Fig, 11.12.

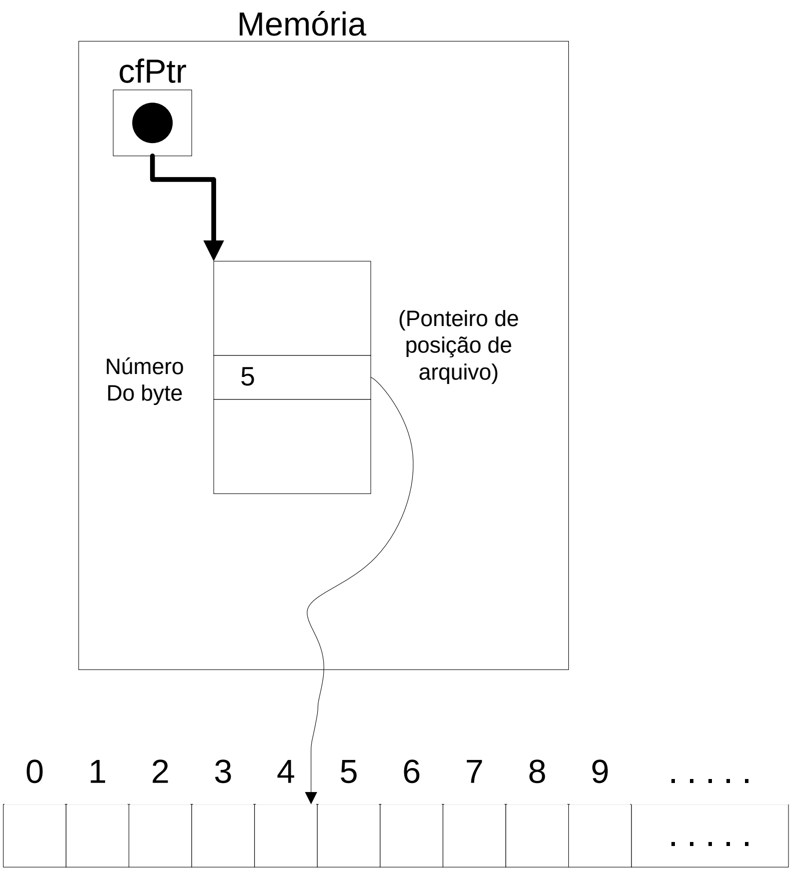


Fig. 11.14 O ponteiro de posição do arquivo indicando um deslocamento (offset) de 5 bytes a partir do ilido do arquivo,

A instrução

fseek(cfPtr, (numConta — 1) *** sizeof(structdadosCliente), SEEK_SET);**

posiciona o ponteiro de posição do arquivo referenciado por **cf Ptr** na localização de bytes calculado por **(numConta - 1) * sizeof (struct dadosCliente)**; o valor dessa expressão é chamado *offset* ou *deslocamento.* Como o número da conta se situa entre 1 e 100 mas as posições de bytes no arquivo iniciam com **0,** é subtraído 1 do número da conta durante o cálculo da localização do byte no registro. Dessa forma, para o registro 1, o ponteiro de posição do arquivo é fixado no byte 0 do arquivo. constante simbólica **SEEK_SET** indica que o ponteiro de posição do arquivo é posicionado em relação ao início do arquivo de acordo com o valor do offset. Como a instrução anterior indica, uma pesquisa pelo número de conta 1 no arquivo fixa o ponteiro de posição do arquivo no início do arquivo porqueo cálculo de localização do byte tem o valor 0. A Fig. 11.14 ilustra o ponteiro de arquivo referente a uma estrutura **FILE** na memória. O ponteiro de posição do arquivo indica que o próximo byte a ser lidi ou gravado está 5 bytes a partir do início do arquivo.

O padrão ANSI mostra o protótipo da função **fseek** como

int fseek(FILE *stream, long int offset, int whence);

em que **offset** é o número de bytes do local **whence** no arquivo apontado por **stream.** O argumento **whence** só pode ter um valor dentre os três valores possíveis **SEEK_SET, SEEK_CUR** ou **SEEK_E^** — indicando a localização no arquivo a partir da qual a pesquisa se inicia. **SEEK_SET** indica que a pesquisa começa no início do arquivo; **SEEK_CUR** indica que a pesquisa começa no local atual no arquivo; e **SEEK_END** indica que a pesquisa inicia no final do arquivo. Essas três constantes simbólicas são definidas no arquivo de cabeçalho **stdio. h.**

11.9 Lendo Dados Aleatoriamente em um Arquivo de Acesso Aleatório

A função **fread** lê um número especificado de bytes de um arquivo para a memória. Por exemploa a instrução

fread(&cliente, sizeof(struct dadosCliente), 1, cfPtr);

lê o número de bytes determinado por **sizeof (struct dadosCliente)** do arquivo refererenciado por **cfPtr** e armazena os dados na estrutura **cliente.** Os bytes são lidos do local no arquivo especificado pelo ponteiro de posição do arquivo. A função **f read** pode ser usada para ler vários elementos, com tamanho fixo, de um array, fornecendo um ponteiro para o array no qual os elementos serão armazenados e indicando o número de elementos a ser lido. A instrução anterior especifica que um elemento deve ser lido. Para ler mais de um elemento, especifique o número de elementos no terceiro argumento da instrução **fread.**

O programa da Fig. 11.15 lê seqüencialmente todos os registros do arquivo **"credito.dat", de**termina se cada registro contém dados e imprime os dados formatados dos registros que contêm dados. A função **feof** determina quando o final do arquivo é alcançado e a função **fread** transfere dados de disco para a estrutura **cliente** do tipo **dadosCliente.**

11.10 Estudo de Caso: Um Programa de Processamento de Transações

Agora apresentamos um programa substancial de processamento de transações usando arquivos de acesso aleatório. O programa gerencia informações de contas bancárias. O programa atualiza contas existentes, acrescenta novas contas, exclui contas e armazena listagens de todas as contas atuais em um arquivo de texto para impressão. Admitimos que o programa da Fig. 11.11 foi executado para criar o arquivo **credito.dat.**

O programa apresenta cinco opções. A opção 1 chama a função **arquivoTexto** para armazenar uma lista formatada de todas as contas em um arquivo de texto chamado **contas.txt** que pode ser impresso posteriormente. A função usa **f read** e as técnicas de arquivo seqüencial empregadas no programa da Fig. 11.15. Depois de escolhida a opção 1, o arquivo **contas.txt** possuirá:

```
/* Lendo seqüencialmente um arquivo de acesso aleatório */
#include <stdio.h>

struct dadosCliente {
        int numConta;
        char sobrenome[15];
        char primNome[10];
        float saldo;
};

main(){

        FILE *cfPtr;
        struct dadosCliente cliente;

        if ((cfPtr = fopen( *credito.dat",   "r"))  == NULL)
                printf("Arquivo nao pode ser aberto.\n");
        else {
                printf("%-6s%-16s%-lls%10s\n","Conta", "Sobrenome","Nome","Saldo");
                while(!feof(cfPtr)){
                        fread(&cliente,sizeof(struct dadosCliente),1,cfPtr);
                        if (cliente.numConta != 0)
                        printf("%-6d%-16s%-lls%10.2f\n",cliente.numConta,
                                cliente.sobrenome,cliente.primNome,cliente.saldo);
                }
        }
        fclose(cfPtr);
        return 0;
}
```

Conta	Sobrenome	Nome	Saldo
29	Brown	Nancy	-24.54
33	Dunn	Stacey	314.33
37	Barker	Doug	0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

Fig. 11.15 Lendo seqüencialmente um arquivo de acesso aleatório.

Conta	**Sobrenome**	**Nome**	**Saldo**
29	**Brown**	**Nancy**	**-24.54**
33	**Dunn**	**Stacey**	**314.33**
37	**Barker**	**Doug**	**0.00**
88	**Smith**	**Dave**	**258.34**
96	**Stone**	**Sam**	**34.98**

A opção 2 chama a função **atualizaRegistro** para atualizar uma conta. A função só atualizará um registro que já exista, portanto a função verifica inicialmente se o registro especificado pelo usuário está vazio. O registro é lido na estrutura **cliente** com **f read** e o membro **numConta** é comparado com 0. Se for 0, o registro não possui informações e é impressa uma mensagem informando que o registro está vazio. A seguir, são exibidas as opções do menu. Se o registro possuir informações, a função **atualizaRegistro** recebe a quantia da transação, calcula o novo saldo e regrava o registro no arquivo. Uma saída típica da opção 2 é:

Digite a conta a atualizar (1 - 100): 37
37 Barker Doug 0.00
Digite debito (+) ou pagamento (-): +87.99
37 Barker Doug 87.99

A opção 3 chama a função **novoRegistro** para adicionar um novo registro ao arquivo. Se o usuário fornecer um número de conta já existente, **novoRegistro** exibirá uma mensagem de erro que o registro já contém informações e as opções do menu serão exibidas novamente. Essa função usa o mesmo processo que o programa da Fig. 11.12 para adicionar uma nova conta. Uma saída típica da opção 3 é:

Digite numero da nova conta (1 - 100): 22 Digite sobrenome, primeiro nome, saldo ? Johnston Sarah 247.45

A opção 4 chama a função **excluiRegistro** para excluir um registro de um arquivo. A exclusão é realizada pedindo ao usuário um número de conta e reinicializando o registro. Se a conta não possuir informações, **excluiRegistro** exibe uma mensagem de erro informando que a conta não existe. A opção 5 termina a execução do programa. O programa é mostrado na Fig. 11.16. Observe que **"credito. dat**" é aberto para atualização (leitura e gravação) usando o modo " **r+".**

```
/* Esse programa le seqüencialmente um arquivo de acesso, *
* aleatório, atualiza os dados ja gravados no arquivo, cria *
* novos dados a serem colocados no arquivo e exclui dados *
* ja presentes no arquivo. */
#include <stdio.h>

struct dadosCliente {
        int numConta;
        char sobrenome[15];
        char primNome[10];
        float saldo;
};

int digitaEscolha(void);
void arquivoTexto(FILE *);
void atualizaRegistro(FILE *);
void novoRegistro(FILE *);
void excluiRegistro(FILE *);

main()
{

FILE *cfPtr;
int escolha;
if  ((cfPtr = fopen( "credito.dat",   "r+"))  == NULL)
        printf("Arquivo nao pode ser aberto.\n"); else {
        while  ((escolha = digitaEscolha())   != 5) {
                switch (escolha) {
                        case 1:
                                arquivoTexto(cfPtr);
                                break;

                        case 2 :
                                atualizaRegistro(cfPtr);
                                break;

                        case 3:
                                novoRegistro(cfPtr);
                                break;

                        case 4:
                                excluiRegistro(cfPtr);
                                break;
                }
        }
fclose(cfPtr);
return 0;
}
```

```
void arquivoTexto(FILE *lePtr) {

FILE *gravaPtr;
struct dadosCliente cliente;
if ((gravaPtr = fopen("contas.txt",  "w")) == NULL)
        printf("Arquivo nao pode ser aberto.\n");
else {
        rewind(lePtr);
        fprintf(gravaPtr,  "%-6s%-16S%-lls%10s\n",
                "Conta",  "Sobrenome",  "Nome",  "Saldo");
        while  (!feof(lePtr)) {
                fread(&cliente,  sizeof(struct dadosCliente),  1,  lePtr);
                if  (cliente.numConta != 0)
                        fprintf(gravaPtr,  "%-6d%-16s%-lls%10.2f\n",
                        cliente.numConta, cliente.sobrenome,
                                cliente.primNome, cliente.saldo);
        }
}
fclose(gravaPtr);
}


void atualizaRegistro(FILE *fPtr) {

int conta;
float transação;
struct dadosCliente cliente;
printf("Digite a conta a atualizar  (1 - 100): ");
scanf("%d", fcconta);
fseek(fPtr,  (conta - 1)  * sizeof(struct dadosCliente), SEEK_SET);
fread(Sccliente,  sizeof(struct dadosCliente),  1,  fPtr);
if  (cliente.numConta == 0)
        printf("Conta #%d nao contem informações.\n", conta);
else {
        printf("%-6d%-16s%-lls%10.2f\n\n",
                cliente.numConta, cliente.sobrenome,
                cliente.primNome, cliente.saldo);
        printf("Digite debito  (+)  ou pagamento  (-): ");
        scanf("%f",  &transacao);
        cliente.saldo += transação;
        printf("%-6d%-16s%-lls%10.2f\n",cliente.numConta,  cliente.sobrenome,
                        cliente.primNome,  cliente.saldo);
        fseek(fPtr,  (conta - 1)  * sizeof(struct dadosCliente), SEEKSET);
        fwrite(&cliente,  sizeof(struct dadosCliente),  1,  fPtr);
}
}
```

```
void excluiRegistro(FILE *fPtr) {

struct dadosCliente cliente,  clienteNulo =  {0,   "",   "", 0};
int numeroConta;
printf("Digite numero da conta a excluir  (1 - 100):   ");
scanf("%d",  &numeroConta);
fseek(fPtr,   (numeroConta - 1)  * sizeof(struct dadosCliente), SEEK_SET);
fread(&cliente,  sizeof(struct dadosCliente),  1, fPtr);
if  (cliente.numConta == 0)
        printf("Conta %d nao existe.\n", numeroConta);
else {
        fseek(fPtr,   (numeroConta - 1)  * sizeof(struct dadosCliente), SEEK_SET);
        fwrite(&clienteNulo,  sizeof(struct dadosCliente),  1,  fPtr);
}
}

void novoRegistro(FILE *fPtr) {

struct dadosCliente cliente; int numeroConta;
printf("Digite numero da nova conta  (1 - 100): ");
scanf("%d",  inumeroConta);
fseek(fPtr,   (numeroConta - 1)  * sizeof(struct dadosCliente), SEEK_SET);
fread(&cliente,  sizeof(struct dadosCliente),  1,  fPtr) ;
if  (cliente.numConta  != 0)
        printf("Conta #%d ja contem informações.\n", cliente.numConta);
else {
        printf("Digite sobrenome, primeiro nome,  saldo\n? ");
        scanf("%s%s%f", &cliente.sobrenome, &cliente.primNome,&cliente.saldo);
        cliente.numConta = numeroConta;
        fseek(fPtr,  (cliente.numConta - 1) *sizeof(struct dadosCliente),SEEK_SET);
        fwrite(&cliente,  sizeof(struct dadosCliente),  1, fPtr);
}
}

int digitaEscolha(void){

int escolhaMenu;
printf("\nDigite sua escolha\n"
        "1 - armazena um arquivo formatado de texto das contas chamado\n"
        "contas.txt \" para impressao\n"
        "2 - atualiza uma conta\n"
        "3 - adiciona uma conta nova\n"
        "4 - exclui uma conta\n"
        "5 - encerra o programa\n?  ");
scanf("%d",  &escolhaMenu);
return escolhaMenu;
}
```

Fig. 11.16 Programa de contabilidade bancária

Resumo

• Todos os itens de dados processados por um computador são reduzidos a combinações de zeros e uns.

• O menor item de dado de um computador pode assumir o valor 0 ou o valor 1. Tal item de dado é chamado um bit (abreviação de "binary digit" ("dígito binário") — um dígito que pode assumir um dentre dois valores).

• Dígitos, letras e símbolos especiais são chamados caracteres. O conjunto de todos os caracteres que podem ser utilizados para escrever programas e representar itens de dados em um determinado computador é chamado conjunto de caracteres daquele computador. Todo caractere do conjunto de caracteres de um computador é representado como um padrão de oito ls e Os (chamado byte).

• Um campo é um grupo de caracteres que possui um significado.

• Um registro é um grupo de campos relacionados entre si.

• Pelo menos um campo de cada registro é escolhido normalmente como chave dos registros. A chave (ou campo-chave) dos registros identifica um registro como pertencendo a uma determinada pessoa ou entidade.

• O tipo mais popular de organização dos registros em um arquivo c chamado arquivo de acesso seqüencial, no qual os registros são acessados consecutivãmente até que os dados desejados sejam localizados.

• Um grupo de arquivos relacionados entre si é chamado algumas vezes banco de dados. Um grupo de programas desenvolvidos para criar e gerenciar bancos de dados é chamado um sistema de gerenciamento de bancos de dados (SGBD, ou database management system, DBMS).

• A linguagem C visualiza cada arquivo simplesmente como um fluxo seqüencial de bytes.

• A linguagem C abre automaticamente três arquivos e seus fluxos associados — entrada padrão, saída padrão e exibição padrão de mensagens de erro (erro padrão) — quando a execução do programa é iniciada.

• Os ponteiros de arquivos atribuídos à entrada padrão, à saída padrão e à exibição padrão de rnensagens de erro são **stdin, stdout** e **stderr,** respectivamente.

• A função **fgetc** lê um caractere de um arquivo específico.

• A função **fpute** grava um caractere em um arquivo específico.

• A função **fgets** lê uma linha de um arquivo específico.

• A função **fputs** grava uma linha em um arquivo específico.

• **FILE** é um tipo de estrutura definido no arquivo de cabeçalho **stdio. h.** O programador não precisa saber os detalhes dessa estrutura para usar arquivos. Quando um arquivo é aberto, é retornado um ponteiro para a estrutura **FILE** do arquivo.

• A função **fopen** utiliza dois argumentos — um nome de arquivo e um modo de abertura de arquivo — e abre o arquivo. Se o arquivo existir, o conteúdo do arquivo é eliminado sem qualquer aviso. Se o arquivo não existir e o arquivo for aberto para gravação, **fopen** cria o arquivo.

• A função **feof** determina se o indicador de fim de arquivo (end-of-file) de um determinado arquivo foi alcançado.

• A função **fprintf** é equivalente a **printf** exceto que **fprintf** recebe como argumento um ponteiro ao arquivo no qual os dados devem ser gravados.

• A função **fclose** fecha o arquivo apontado por seu argumento.

• Para criar um arquivo ou para eliminar o conteúdo dc um arquivo antes de gravar dados, abra o arquivo para gravação ("w"). Para ler um arquivo existente, abra-o para leitura (" **r**"). Para adicionar registros ao final de um arquivo existente, abra o

arquivo para anexação (**"a"**). Para abrir um arquivo de forma que seja possível nele gravar dados e lê-lo, abra o arquivo para atualização com um dentre os três modos de atualização existentes — **" r+", " w+**" ou " **a+**". O modo " **r+**" simplesmente abre o arquivo para leitura e gravação. O modo " w+" cria o arquivo se ele não existir e elimina o conteúdo atual do arquivo se ele existir. O modo " **a+**" cria o arquivo se ele não existir e a gravação é feita no final do arquivo.

• A função **fscanf** é equivalente a **scanf** exceto que **fscanf** recebe como argumento um ponteiro para o arquivo (normalmente diferente de **stdin)** do qual os dados serão lidos.

• A função **rewind** faz com que o programa reposicione o ponteiro de posição do arquivo especificado para o início do arquivo.

• O processamento de arquivos aleatórios é usado para ter acesso direto a um registro.

• Para facilitar o acesso aleatório, os dados são armazenados em registros de comprimento fixo. Como todos os registros são de mesmo comprimento, o computador pode calcular rapidamente (em função do campo-chave dos registros) a localização exata de um registro em relação ao início do arquivo.

• Os dados podem ser acrescentados facilmente a um arquivo de acesso aleatório sem destruir os outros dados no arquivo. Além disso, os dados previamente armazenados em um arquivo com registros de comprimento fixo podem ser modificados e excluídos sem que haja necessidade de regravar o arquivo inteiro.

• A função **fwrite** grava um bloco (número específico de bytes) de dados em um arquivo.

• O operador de tempo de compilação **sizeof** retorna o tamanho em bytes de seu operando.

• A função **fseek** define o ponteiro de posição de arquivo em uma posição específica de um arquivo baseada no local inicial de pesquisa no arquivo. A pesquisa pode iniciar em uma de três posições possíveis — **SEEK_SET** inicia no início do arquivo, **SEEK_CUR** inicia na posição atual no arquivo e **SEEK_END** inicia no fim do arquivo.

• A função **fread** lê um bloco (número específico de bytes) de dados de um arquivo.

Terminologia

abrir um arquivo
acesso aleatório
arquivo
arquivo de acesso aleatório
arquivo de acesso seqüencial
banco de dados
binario digit
buffer de arquivos
Byte
campo
campo alfabético
campo alfanumérico
campo de caracteres
campo numérico
caractere
chave dos registros
definição de caractere
deslocamento
dígito binário
digito decimal
end-of-file
entrada/saída formatada
espaços finais
espaços iniciais
estrutura
FILE fclose
fechar um arquivo **feof**
fgetc
fgets
fim de arquivo
fopen
fprintf
fpute
fputs
fread
fscanf
fseek
fwrite
hierarquia de dados
indicador de fim de arquivo
letra
modo **a** de abertura de arquivo
modo **a+** de abertura de arquivo
modo de abertura de arquivo
modo **r** de abertura de arquivo
modo **r+** de abertura de arquivo
modo **w** de abertura de arquivo
modo **w+** de abertura de arquivo
nome de arquivo
número de dupla precisão
número de precisão simples
número inteiro offset
ordem alfa
parâmetro de número de registros
ponteiro de arquivo
ponteiro de posição de arquivo
registro
rewind
SEEK_CUR
SEEK_END
SEEK_SET
sistema de gerenciamento de banco de dados
stderr (exibição padrão de mensagens de erro)
stdin (entrada padrão)
stdout (saída padrão) zeros e uns

Erros Comuns de Programação

11.1 Abrir para gravação (" **w**") um arquivo existente quando, na realidade, o usuário desejava preservar o arquivo: o conteúdo do arquivo é eliminado sem qualquer aviso.

11.2 Esquecer-se de abrir um arquivo antes de tentar fazer referência a ele em um programa.

11.3 Usar o ponteiro errado de arquivo para fazer referência a um arquivo.

11.4 Abrir para leitura um arquivo não-existente.

11.5 Abrir para leitura ou gravação um arquivo sem ter garantido os direitos apropriados de acesso em relação ao arquivo.

11.6 Abrir para gravação um arquivo quando não houver espaço disponível em disco. A Fig. 11.6 lista os modos de abrir arquivos.

11.7 Abrir um arquivo com o modo incorreto pode levar a erros terríveis. Por exemplo, abrir um arquivo no modo de gravação (" w") quando ele deve ser aberto no modo de atualização (" r +") faz com que o conteúdo do arquivo seja eliminado.

Práticas Recomendáveis de Programação

11.1 Certifique-se de que as chamadas a funções de processamento de arquivo em um programa contêm os ponteiros corretos de arquivos.

11.2 Feche explicitamente os arquivos tão logo saiba que o programa não fará nova referência a eles.

11.3 Abrir um arquivo apenas para leitura (e não atualização) se seu conteúdo não deve ser modificado. Isso evita modificações não-intencionais do conteúdo do arquivo. Isso é outro exemplo do princípio do privilégio mínimo.

Dicas de Performance

11.1 Fechar um arquivo pode liberar recursos pelos quais outros usuários podem estar esperando.

11.2 Muitos programadores pensam erradamente que **sizeof** é uma função e que usá-lo gera o overhead de tempo de execução da chamada de uma função. Não existe tal overhead porque **sizeof** é um operador de tempo de compilação.

Dica de Portabilidade

11.1 A estrutura **FILE** varia conforme o sistema operacional (i.e., os membros da estrutura variam entre sistemas conforme o modo como cada sistema manipula seus arquivos).

Exercícios de Revisão

11.1 Preencha as lacunas de cada uma das sentenças a seguir:

a) Em última análise, todos os itens de dados processados por um computador são reduzidos a combinações de_e_.

b) O menor item de dado que um computador pode processar é chamado_.

c) Um_é um grupo de registros relacionados entre si.

d) Dígitos, letras e símbolos especiais são chamados_.

e) Um grupo de arquivos relacionados entre si é chamado_.

f) A função_fecha um arquivo.

g) A instrução_lê dados de um arquivo, de forma similar ao modo como **scanf** lê de **stdin.**

h) A função_lê um caractere de um arquivo especificado.

i) A função_lê uma linha de um arquivo especificado.

j) A função_abre um arquivo.

k) A função_é usada normalmente na leitura de dados de um arquivo em aplicações de acesso. aleatório.

1) A função reposiciona o ponteiro de posição do arquivo em um local específico do arquivo.

11.2 Diga se as afirmações a seguir são verdadeiras ou falsas (para as afirmações falsas, explique o motivo).

a) A função **f scanf** não pode ser usada para ler dados da entrada padrão.

b) O programador deve usar explicitamente **fopen** para abrir os fluxos de entrada padrão, saída padrão e de exibição padrão de mensagens de erro.

c) Um programa deve chamar explicitamente a função **f close** para fechar um arquivo.

d) Se o ponteiro de posição do arquivo apontar, em um arquivo seqüencial, para um local diferente do ínicio do arquivo, esse deve ser fechado e reaberto para leitura a partir do início do arquivo.

e) A função **fprintf** pode escrever na saída padrão.

f) Os dados em arquivos de acesso seqüencial são sempre atualizados sem gravação de outros dados.

g) Não é necessário pesquisar todos os registros em um arquivo de acesso aleatório para encontrar um registro específico.

h) Os registros em arquivos de acesso aleatório não possuem comprimento uniforme.

i) A função **fseek** só pode realizar pesquisas relativas ao início do arquivo.

11.3 Escreva uma instrução simples para realizar cada uma das seguintes tarefas. Admita que todas essas instruções se aplicam ao mesmo programa.

a) Escreva uma instrução que abra o$_u$arquivo " **oldmast. dat**" para leitura e atribua a **of Ptr** o ponteiro de arquivo que retornar.

b) Escreva uma instrução que abra o arquivo " **trans.dat**" para leitura e atribua a **tf Ptr** o ponteiro de arquivo que retornar.

c) Escreva uma instrução que abra o arquivo " **newmast.dat**" para gravação (e criação) e atribua a **nfPtr** o ponteiro de arquivo que retornar.

d) Escreva uma instrução que leia um registro do arquivo **"oldmast.dat**". O registro consiste no inteiro **numConta,** da string **nome** e do ponto flutuante **saldoAtual.**

e) Escreva uma instrução que leia um registro do arquivo " **trans.dat**". O registro consiste no inteiro **numConta** e do ponto flutuante **quantiaReais.**

f) Escreva uma instrução que grave um registro no arquivo "**newmast.dat**". O registro consiste no inteiro **numConta,** da string **nome** e do ponto flutuante **saldoAtual.**

11.4 Encontre o erro em cada um dos seguintes segmentos de programa. Explique como o erro pode ser corrigido.

a) O arquivo referenciado por **f Ptr ("pagar. dat")** não foi aberto.
fprintf(fPtr, *"%ã%s%ã\n",* **conta, companhia, quantia);**

b) **open("receber.dat", "r+");**

c) A instrução a seguir deve ler um registro do arquivo **"pagar. dat**". O ponteiro **pagPtr** se refere a esse arquivo e o ponteiro **recPtr** se refere ao arquivo " **receber. dat".**
fscanf(recPtr, "%d%s%d\n", fcconta, companhia, fcquantia);

d) O arquivo " **ob j etos . dat**" deve ser aberto para inclusão de dados no arquivo sem eliminação dos dados atuais.
if ((tfPtr = fopenCobjetos.dat", "w")) != NULL)

e) O arquivo " **cursos. dat**" deve ser aberto para anexação sem modificação do conteúdo atual do arquivo.
if ((cfPtr = fopenCcursos.dat", "w+")) != NULL)

Respostas dos Exercícios de Revisão

11.1 a) ls, Os. b) Bit. c) Arquivo, d) Caracteres, e) Banco de dados, f) **f close.** g) **f scanf**. h) **getc** ou **f getc.** i) **fgets.** j) **fopen.** k) **f read.** 1) **f seek.**

11.2 a) Falso. A função **f scanf** pode ser usada para ler da entrada padrão incluindo o ponteiro para o fluxo da entrada padrão, **stdin,** na chamada a **f scanf.**

b) Falso. Esses três fluxos são abertos automaticamente pelo C no início da execução do programa.

c) Falso. Os arquivos serão fechados quando a execução do programa termina, mas todos os arquivos devem ser fechados explicitamente com **f close.**

d) Falso. A função **rewind** pode ser usada para reposicionar o ponteiro de posição do arquivo no início do arquivo.

e) Verdadeiro.

f) Falso. Na maioria dos casos, os registros de arquivos seqüenciais não possuem comprimento uniforme. Portanto, é possível que a atualização de um registro faça com que outros dados sejam sobrescritos.

g) Verdadeiro.

h) Falso. Normalmente os registros de um arquivo de acesso aleatório possuem comprimento uniforme.

i) Falso. E possível pesquisar a partir do início do arquivo, a partir do final do arquivo e a partir da posição atual no arquivo de acordo com o local do ponteiro de posição do arquivo.

11.3 a) **ofPtr = fopen("oldmast.dat", "r");**
b) **tfPtr = fopen("trans.dat", "r");**
c) **nfPtr = fopen("newmast.dat", "w");**
d) **fscanf(ofPtr,** *"%â%s%£",* **SnumConta, nome, &saldoAtual);**
e) **fscanf(tfPtr,** *"%â%£",* **&numConta, &quantiaReais) ;**
fj **fprintf(nfPtr,** *"%ã%s%.2£",* **numConta, nome, saldoAtual);**

11.4 a) Erro: O arquivo **"pagar. dat**" não foi aberto antes da referência ao seu ponteiro.
Correção: Use **f open** para abrir **"pagar. dat**" para gravação, anexação ou atualização.
b) Erro: A função **open** não é uma função do ANSI C. Correção: Use a função **f open.**
c) Erro: A instrução **f scanf** usa o ponteiro incorreto de arquivo para fazer referência ao arquivo **"pagar.dat".**
Correção: Use o ponteiro de arquivo **pagPtr** para fazer referência a **"pagar. dat" .**

d) Erro: O conteúdo do arquivo é eliminado porque o arquivo é aberto para gravação ("**w").** Correção: Para adicionar dados ao arquivo, ou abra o arquivo para atualização (" **r** +") ou abra o arquivo para anexação (" **a").**
e) Erro: O arquivo " **cursos . dat**" é aberto para atualização no modo " **w+**" que elimina o conteúdo atual
Correção: Abra o arquivo no modo " **a".**

Exercícios

11.5 Preencha as lacunas de cada uma das sentenças a seguir:
a) Os computadores armazenam grandes quantidades de dados em dispositivos de memória secundária como
b) Um_é composto de vários campos.
c) Um campo que pode conter dígitos, letras e espaços em branco é chamado um campo_
d) Para facilitar a recuperação de registros específicos de um arquivo, um campo de cada registro é escolhido como um_.
e) A grande maioria das informações armazenadas em sistemas computacionais está contida em arquivos
f) Um grupo de caracteres relacionados entre si que possuem um significado é chamado_
g) Os ponteiros de arquivos para os três arquivos abertos automaticamente pelo C quando é iniciada a execução do programa são chamados_,_e_.
h) A função_grava um caractere em um arquivo especificado.
i) A função_grava uma linha em um arquivo especificado.
j) A função_é usada geralmente para gravar dados em um arquivo de acesso aleatório.
k) A função_reposiciona o ponteiro de posição do arquivo no início do arquivo.

11.6 Diga se cada uma das sentenças a seguir é verdadeira ou falsa (para as que forem falsas, explique o motivo).
a) As funções impressionantes realizadas pelos computadores envolvem essencialmente a manipulação de zeros e uns.
b) As pessoas preferem manipular bits em vez de caracteres e campos porque os bits são mais compactos.
c) As pessoas especificam programas e itens de dados como caracteres; a seguir, os computadores manipuIam e processam esses caracteres como grupos de zeros e uns.
d) O código postal de uma pessoa é um exemplo de campo numérico.
e) O endereço residencial de uma pessoa é considerado geralmente um campo alfabético em aplicações computacionais.
f) Os itens de dados processados por um computador formam uma hierarquia de dados na qual aqueles itens se tornam maiores e mais complexos à medida que nos dirigimos de campos para caracteres e para bits etc.
g) Um campo-chave de registros identifica um registro como pertencente a um determinado campo.

h) A maioria das organizações armazena suas informações em um único arquivo para facilitar o processamento computacional.

i) Em programas na linguagem C, os arquivos são sempre chamados pelo nome.

j) Quando um programa cria um arquivo, esse é mantido automaticamente pelo computador para futura referência.

11.7 O Exercício 11.3 pediu ao leitor que escrevesse uma série de instruções simples. Na realidade, essas instruções formam o núcleo de um tipo importante de programa de processamento de arquivos, que é um programa de correspondência de arquivos. No processamento comercial, é comum haver vários arquivos em cada sistema. Em um sistema contábil de contas a receber, por exemplo, geralmente há um arquivo-mestre contendo informações detalhadas sobre cada cliente como seu nome, endereço, número do telefone, saldo a pagar limite de crédito, condições de desconto, cláusulas de contratos e possivelmente um histórico condensado de compras e pagamentos recentes.

À medida que as transações ocorrem (i.e., são feitas compras e realizados pagamentos na conta), elas são registradas no arquivo. Ao final de cada período comercial (i.e., um mês para algumas companhias, uma semana para outras, um dia em ainda outros casos) o arquivo de transações (chamado " **trans . dat"** no Exercício 11.3) é aplicado ao arquivo-mestre (chamado **"oldmast.dat"** no Exercício 11.3), atualizando assim o registro de pagamentos e compras de cada conta. Depois de cada uma dessas atualizações ser realizada, o arquivo-mestre é gravado novamente como um novo arquivo (" **newmast. dat"),** que então é usado no final do próximo período comercial para iniciar mais uma vez o processo de atualização.

Os programas de correspondência de arquivos devem lidar com determinados problemas que não existem em programas de arquivos simples. Por exemplo, nem sempre ocorre uma correspondência. Um cliente em um arquivo-mestre pode não ter feito nenhuma compra ou nenhum pagamento no período comercial atual e portanto não aparecerá no arquivo de transação nenhum registro para esse cliente. Similarmente, um cliente que fez algumas compras ou alguns pagamentos pode ter acabado de se mudar para essa comunidade e a companhia ainda não teve a oportunidade de criar um registro-mestre para esse cliente.

Use as instruções escritas no Exercício 11.3 como base para escrever um programa completo de correspondência de arquivos de contas a receber. Use o número da conta em cada arquivo como campo-chave dos registros para as correspondências. Assuma que cada arquivo é um arquivo seqüencial com registros armazenados na ordem crescente do número de contas.

Quando acontecer uma correspondência (i.e., quando aparecerem registros com o mesmo número de conta tanto no arquivo-mestre como no arquivo de transações), adicione ao saldo atual no arquivo-mestre a quantia em reais do arquivo de transações e grave o arquivo " **newmast. dat**". (Suponha que as compras são indicadas por quantias positivas no arquivo de transações e os pagamentos são indicados por quantias negativas.) Quando houver um registro-mestre de uma determinada conta, mas não houver registro de transação correspondente, simplesmente grave o registro-mestre em " **newmast. dat**". Quando houver uma transação mas não houver um registro-mestre, imprima a mensagem **"Nao ha correspondência entre o registro de transação e o numero da conta** ..." (preencha o número da conta do registro da transação).

11.8 Depois de escrever o Exercício 11.7, escreva um programa simples para criar alguns dados de teste para verificar o programa do Exercício 11.7. Use a seguinte amostra de dados de contabilidade.

Arquivo-mestre: Número da conta	Nome	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Arquivo de Transações: Número da Conta	Quantia em reais
100	27.14
300	62.11
400	100.56
900	82.17

11.9 Execute o programa do Exercício 11.7 usando os arquivos de dados de teste criados no Exercício 11.8. Use a listagem de programa da Seção 11.7 para imprimir um novo arquivo-mestre. Examine cuidadosamente os resultados.

11.10 É possível (na verdade, é comum) ter vários registros de transações com a mesma chave de registros. Isso ocorre porque um cliente em particular pode fazer várias compras e vários pagamentos durante um período comercial. Escreva novamente seu programa de correspondência de arquivos de contas a receber, do Exercício 11.7 para oferecer a possibilidade de manipular vários registros de transação com a mesma chave de registros. Modifique os dados de teste do Exercício 11.8 para incluir os seguintes registros de transações adicionais:

Número da conta	Quantia em reais
300	83.89
700	80.78
700	01.53

11.11 Escreva instruções que realizem cada um dos pedidos a seguir. Assuma que a estrutura

```
struct pessoa {
char sobrenome[15]; char primNome[15]; char idade[2];
};
```

foi definida e que o arquivo já está aberto para gravação.

a) Inicialize o arquivo **"nomeidad.dat"** de forma que haja 100 registros com **sobrenome = "unassigned", primNome** = "" e**idade = "0".**

b) Digite 10 sobrenomes, primeiros nomes e idades e grave-os no arquivo.

c) Atualize um registro se não houver informações no registro, diga ao usuário " **Sem info".**
d) Exclua um registro que não tenha informações reinicializando aquele registro.

11.12 Você é o proprietário de uma loja de ferragens e precisa conservar um inventário que informe que ferramentas você possui, quantas existem e o custo de cada uma. Escreva um programa que inicialize o arquivo **"ferragem.dat"** com 100 registros vazios, permita a entrada dos dados referentes a cada ferramenta permita a criação de uma listagem de todas as ferramentas, permita a exclusão de um registro de uma ferramenta que você não possua mais e permita a atualização de *qualquer* informação do arquivo. O número de identificação da ferramenta deve ser o campo-chave dos registros. Use as seguintes informações para iniciar seu arquivo.

Registro #	Nome da ferramenta	Quantidade	Custo
3	Lixa elétrica	7	57.98
17	Martelo	76	11.99
24	Serra tico-tico	21	11.00
39	Cortador de grama	3	79.50
56	Serra elétrica	18	99.99
68	Chave de fenda	106	6.99
77	Marreta	11	21.50
83	Chave inglesa	34	7.50

11.13 **Gerador de Caracteres Alfanuméricos para Números de Telefone.** Os teclados normais de telefones possuem os dígitos de 0 a 9. Do número 2 ao 9, cada número está associado a três letras, como indica a tabela a seguir.

Dígito	Letra
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Muitas pessoas acham difícil memorizar números de telefone, portanto elas usam a correspondência entre dígitos e letras para desenvolver palavras de sete letras que correspondam a seus números de telefone. Por exemplo, uma pessoa cujo número de telefone seja 686-3767 poderia usar a correspondência indicada na tabela acima para desenvolver a palavra de sete letras "NÚMEROS".

Freqüentemente, as empresas tentam obter números de telefone que sejam fáceis para seus clientes lembrarem. Se uma empresa pode fazer propaganda de uma única palavra para seus clientes discarem, sem dúvida a empresa receberá algumas chamadas a mais.

Cada palavra de sete letras corresponde a exatamente um número de telefone de sete dígitos. O restaurante que desejasse aumentar seu serviço de entrega a domicílio poderia seguramente fazer isso com o número 368-7342 (i.e., "ENTREGA").

Cada número de sete dígitos corresponde a muitas palavras diferentes de sete letras. Infelizmente, a maioria delas representa justaposições de letras sem qualquer significado. Entretanto, é possível que o proprietário de uma barbearia ficasse satisfeito de saber que o número de telefone de seu estabelecimento, 2223567, corresponde a "CABELOS". O proprietário de uma loja de bebidas, sem dúvida alguma, ficaria radiante se descobrisse que o número de telefone de sua loja, 232-4327, corresponde a "BEBIDAS". Um veterinário com o número de telefone 264-6247 ficaria contente de saber que o número corresponde às letras "ANIMAIS".

Escreva um programa em C que, dado um número de sete dígitos, grave em um arquivo todas as combinações possíveis de palavras de sete letras que correspondem a cada número. Há 2187 (3 elevado à sétima potência) de tais palavras. Evite números de telefone com os dígitos 0 e 1.

11.14 Se você tem um dicionário computadorizado disponível, modifique o programa escrito no Exercício 11.13 para verificar as palavras no dicionário. Algumas combinações de sete letras criadas por esse programa consistem em duas ou mais palavras (o número de telefone 848-2236 produz "VIVABEM").

11.15 Modifique o exemplo da Fig. 8.14 para usar as funções **f getc** e **fputs** em vez de **getchar** e **puts.** O programa deve oferecer ao usuário a opção de ler da entrada padrão e escrever na saída padrão ou de ler em um arquivo especificado e gravar em outro arquivo especificado. Se o usuário escolher a segunda opção, precisará fornecer os nomes dos arquivos para entrada e saída.

11.16 Escreva um programa que use o operador **sizeof** para determinar os tamanhos em bytes dos vários tipos de dados em seu sistema computacional. Escreva os resultados no arquivo " **tamdados . dat**" para que possam ser impressos mais tarde. O formato dos resultados no arquivo deve ser:

Tipo de dado Tamanho
char 1
unsigned char 1
short int 2
unsigned short int 2
int 4
unsigned int 4
long int 4
unsigned long int 4
float 4
double 8
long double 16

Nota: Os tamanhos dos tipos em seu computador podem não ser os mesmos que os listados anteriormente.

11.17 No Exercício 7.19, você escreveu uma simulação de software de computador que usou uma linguagem de máquina especial chamada Linguagem de Máquina Simpletron (LMS). Na simulação, cada vez que você quisesse rodar um programa LMS, digitava o programa no simulador. Se você cometesse um erro ao digitar o programa LMS, o simulador era reiniciado e o código LMS era digitado novamente. Seria bom ter a capacidade de ler um programa LMS de um arquivo em vez de digitá-lo todas as vezes. Isso reduziria o tempo e os erros na preparação da execução de programas LMS.

a) Modifique o simulador escrito no Exercício 7.19 para ler programas LMS de um arquivo especificado pelo usuário no teclado.

b) Depois de o Simpletron ser executado, ele imprime na tela o conteúdo de seus registros e memória. Seria bom capturar a saída do arquivo, sendo assim modifique o simulador para imprimir sua saída em um arquivo além de exibir a saída na tela.

As estruturas são usadas normalmente para definir registros a serem armazenados em arquivos (veja o Capítulo 11, "Processamento de Arquivos"). Os ponteiros e as estruturas facilitam a formação de estruturas mais complexas de dados como listas encadeadas, filas (queues), pilhas (stacks) e árvores (veja o Capítulo 12, "Estruturas de Dados").

12

Estruturas de Dados

Objetivos

• Ser capaz de alocar e liberar memória dinamicamente para objetos de dados.

• Ser capaz de formar estruturas encadeadas de dados usando ponteiros, estruturas referenciadas e recursão.

• Ser capaz de criar e manipular listas encadeadas, filas, pilhas e árvores binárias.

• Entender várias aplicações importantes de estruturas encadeadas de dados.

Muito do que eu tinha, não consegui libertar;
Muito do que eu libertei voltou para mim.
Lee Wilson Dodd

" Você poderia andar mais depressa? " disse um peixe para um caracol. "
" Há um golfinho atrás de nós e ele está muito perto. "
Lewis Carroll

Há sempre um lugar no topo.
Daniel Webster

Vá em frente — continue andando.
Thomas Morton

Acho que nunca verei
Um poema tão encantador quanto uma árvore.
Joyce Kilmer

Sumário

12.1 Introdução

Estudamos *estruturas de dados* de tamanho fixo como arrays unidimensionais, arrays bidimensionais e structs. Este capítulo apresenta as *estruturas dinâmicas de dados* com tamanhos que aumentam e diminuem durante a execução de um programa. As *listas encadeadas* são grupos de itens de dados "colocados em linha" — as inserções e exclusões são feitas em qualquer lugar de uma lista encadeada. As *pilhas (stacks)* são importantes em compiladores e sistemas operacionais — as inserções e exclusões são feitas apenas em uma extremidade de uma pilha — seu *topo.* As *filas (queues)* representam filas de espera; as inserções são feitas na extremidade posterior (também chamada *fim* ou *cauda)* de uma fila e as exclusões são feitas na extremidade anterior (também chamada *início)* de uma fila. As *árvores binárias* facilitam a pesquisa rápida e a classificação dos dados, a eliminação eficiente de itens duplicados de dados, a representação de diretórios de sistemas de arquivos e a compilação de expressões em linguagem de máquina. Cada uma dessas estruturas de dados tem muitas outras aplicações interessantes.

Analisaremos cada um dos tipos principais das estruturas de dados e implementaremos programas que criam e manipulam essas estruturas.

Este capítulo é complexo e fascinante. Os programas são completos e incorporam a maioria do que você aprendeu nos capítulos anteriores. Os programas se concentram especialmente na manipulação de ponteiros, um assunto que muitas pessoas consideram estar entre os tópicos mais difíceis da linguagem C. O capítulo está repleto de programas altamente práticos que você poderá usar em cursos mais avançados; inclui também um ótimo conjunto de exercícios que enfatizam as aplicações práticas das estrutuas de dados.

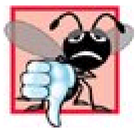
Desejamos sinceramente que você tente resolver o projeto mais importante descrito na seção especial intitulada "Construindo Seu Próprio Compilador". Você usou um compilador para traduzir seus programas na linguagem C para linguagem de máquina, para que seus programas pudessem ser executados no computador. Nesse projeto, você construirá realmente seu próprio compilador. Ele lerá um arquivo de instruções escritas em uma linguagem de alto nível simples, mas poderosa, similar às primeiras versões do BASIC. Seu compilador traduzirá essas instruções para um arquivo de instruções da Linguagem de Máquina Simpletron. A LMS é a linguagem que você aprendeu na seção especial do Capítulo 7, "Construindo Seu Próprio Computador". A seguir, seu programa Simulador Simpletron executará o programa LMS produzido por seu compilador! Esse projeto lhe fornecerá uma excelente oportunidade para exercitar a maior parte do que você aprendeu neste curso. A seção especial lhe conduz cuidadosamente através das especificações das linguagens de alto nível e descreve os algoritmos necessários para converter cada tipo de instrução de linguagem de alto nível em instruções de linguagem de máquina. Se você gosta de desafios, pode tentar solucionar as muitas questões para aperfeiçoar tanto o compilador como o Simulador Simpletron sugeridas nos Exercícios.

12.2 Estruturas Auto-referenciadas

Uma *estrutura auto-referenciada* contém um membro ponteiro que aponta para uma estrutura do mesmo tipo. Por exemplo, a definição

struct node { int data;
struct node *nextPtr;

define um tipo, **struct node.** Uma estrutura do tipo **struct node** tem dois membros — um membro inteiro **data** e um membro ponteiro **nextPtr.** O membro **nextPtr** aponta para uma estrutura do tipo **struct node** — uma estrutura do mesmo tipo que o da declarada aqui, daí o termo "estrutura auto-referenciada". O membro **nextPtr** é chamado *link* — i.e., **nextPtr** pode ser usado para "ligar" uma estrutura do tipo **struct node** com outra estrutura do mesmo tipo. As estruturas auto-**referenciadas** podem ser ligadas **entre** si para formar estruturas úteis de dados como listas, filas, pilhas e árvores. A Fig. 12.1 ilustra duas estruturas auto-referenciadas ligadas entre si para formar uma lista. Observe que uma barra — representando um ponteiro **NULL** — está colocada no membro de ligação da segunda estrutura auto-referenciada para indicar que o link não aponta para outra estrutura. A barra serve apenas para ilustração, ela não significa o caractere correspondente em C. Normalmente um ponteiro **NULL** indica o final de uma estrutura de dados da mesma forma que o caractere **NULL** indica o final de uma string.



Erro comun de programação 12.1

*Não definir como **NULL** o link no último nó de uma lista.*

12.3 Alocação Dinâmica da Memória

Criar e manter estruturas dinâmicas de dados exige *alocação dinâmica de memória* — a capacidade de um programa obter, em tempo de execução, mais espaço de memória para conter novos nós e liberar espaço não mais necessário. O limite máximo da alocação dinâmica de memória pode ser a quantidade **de** memória física disponível **no** computador **ou** a quantidade de memória virtual disponível em um sistema **de** memória virtual. Freqüentemente, **os** limites são muito menores porque a memória dispor: s deve ser compartilhada por muitos usuários.

As funções **malloc** e **free** e o operador **sizeof** são essenciais para a alocação dinâmica da memória. A função **malloc** utiliza como argumento o número **de** bytes a serem alocados e retorna um ponteiro **do** tipo **void*** *{ponteiro para void)* para a memória alocada. Um ponteiro **void*** pode ser atribuído a uma variável **de** qualquer tipo **de** ponteiro. A função **malloc** é usada normalmente com o operador **sizeof.** Por exemplo, a instrução

newPtr = malloc(sizeof(struct node));

processa **sizeof (struct node)** para determinar o tamanho em bytes **de** uma estrutura do tipo **struct node,** aloca uma nova área **de sizeof (struct node)** na memória e armazena na variável **newPtr** um ponteiro para a memória alocada. Se não houver memória disponível, **malloc** retorna um ponteiro **NULL.**

A função **free** libera memória alocada — i.e., a memória é retornada ao sistema para que possa ser realocada **no** futuro. Para liberar memória alocada dinamicamente pela chamada precedente a malloc use a instrução

free(newPtr);

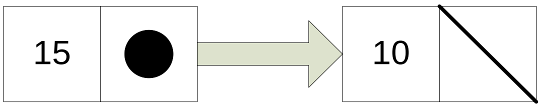
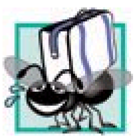


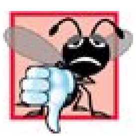
Fig. 12.1 Duas estruturas auto-referenciadas ligadas entre si.

As seções a seguir analisam listas, pilhas, filas e árvores. Cada uma dessas estruturas de dados é criada e mantida com alocação dinâmica de memória e estruturas auto-referenciadas.



Dicas de portabilidade 12.1

O tamanho de uma estrutura não é necessariamente a soma dos tamanhos de seus membros. Isso acontece devido às várias exigências de alinhamento de limites que diferem de um equipamento para outro (veja o Capítulo 10)



Erro comun de programação 12.2

Admitir que o tamanho de uma estrutura é simplesmente a soma dos tamanhos de seus membros.

Boa prática de programação 12.1

*Use o operador **sizeof** para determinar o tamanho de uma estrutura.*

Boa prática de programação 12.2

*Ao usar **malloc,** faça a verificação se o valor de retorno do ponteiro é **NULL.** Imprima uma mensagem de erro se a memória solicitada não foi alocada.*

Erro comun de programação 12.3

Não liberar memória alocada dinamicamente quando ela não mais for necessária pode fazer com que o sistema esgote prematuramente sua memória. Algumas vezes isso é chamado um "vazamento de memória".

Boa prática de programação 12.3

*Quando a memória que foi alocada dinamicamente não for mais necessária, use **free** para devolver imediatamente a memória ao sistema.*

Erro comun de programação 12.4

*Liberar memória não alocada dinamicamente com **malloc.***

Erro comun de programação

Fazer referência à memória que foi liberada.

12.4 Listas Encadeadas

Uma *lista encadeada* (ou *lista linear)* é um conjunto linear de estruturas auto-referenciadas, chamadas nós, conectadas por *links (ligações* ou *encadeamento)* de ponteiros — daí o termo lista "encadeada" ("linked" list, em inglês). Uma lista encadeada é acessada por meio de um ponteiro para o primeiro nó da lista. Os nós subseqüentes são acessados por meio do membro ponteiro de ligação (link) armazenado em cada nó. Os dados são armazenados dinamicamente em uma lista encadeada — cada nó é criado quand necessário. Um nó pode conter dados de qualquer tipo, incluindo outras structs. Pilhas e filas também podem ser estruturas lineares e, como veremos, são versões limitadas das listas encadeadas. As árvores são estruturas não-lineares de dados.

As listas de dados podem ser armazenadas em arrays, mas as listas encadeadas apresentam várias vantagens. Uma lista encadeada é recomendável quando o número de elementos de dados a serem representados na estrutura de dados não pode ser previsto imediatamente. As listas encadeadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir quando necessário. Entretanto, o tamanho de um array não pode ser alterado porque a memória do array é alocada em tempo de compilação. Os arrays podem ficar cheios. As listas encadeadas ficarão cheias quando o sistema não tiver memória suficiente para satisfazer as exigências de alocação dinâmica do armazenamento.

Dica de desempenho 12.1

Um array pode ser declarado deforma a conter mais elementos do que o número esperado de itens de dados, mas isso pode desperdiçar memória. As listas encadeadas permitem utilização melhor da memória nessas situações.

As listas encadeadas podem ser mantidas segundo uma classificação ordenada inserindo cada elemento novo no local adequado da lista.

Dica de desempenho 12.2

A inserção e a eliminação (remoção) em um array ordenado podem ser demoradas — todos os elementos a partir do elemento inserido ou removido devem ser deslocados adequadamente.

Dica de desempenho 12.3

Os elementos de um array podem ser armazenados contiguamente na memória. Isso permite acesso imediato a qualquer elemento do array porque o endereço de qualquer elemento pode ser calculado diretamente com base em sua posição relativa ao início do array. As listas encadeadas não permitem tal acesso imediato aos seus elementos.

Normalmente os nós de listas encadeadas não são armazenados contiguamente na memória. Entretanto, logicamente os nós de uma lista encadeada parecem estar contíguos. A Fig. 12.2 ilustra uma encadeada com vários nós.

Dica de desempenho 12.4

Usar alocação dinâmica de memória (em vez de arrays) para estruturas de dados que aumentam e diminuem em tempo de execução pode economizar memória. Entretanto, tenha em mente que os ponteiros ocupam espaço e que a memória alocada dinamicamente oferece o risco de overhead de chamadas de funções.

O programa da Fig. 12.3 (a saída do programa está mostrada na Fig. 12.4) manipula uma lista de caracteres. O programa fornece duas opções: 1) inserir na ordem alfabética um caractere na lista (função **insert** e 2) remover um caractere da lista (função **delete).** Esse é um programa grande e complexo. Segue-se uma análise detalhada do programa. O Exercício 12.20 pede ao aluno que seja implementada uma função recursiva que imprima uma lista de trás para frente. O Exercício 12.21 pede ao aluno que seja implementado uma função recursiva que procure um determinado item de dados em uma lista encadeada.

As duas funções principais das listas encadeadas são **insert** e **delete.** A função **isEmpty é** chamada *função predicada* — ela não altera a lista de nenhuma maneira; na verdade ela determina se a lista está vazia (i.e., se o ponteiro para o primeiro nó da lista é **NULL).** Se a lista estiver vazia, 1 é retornado; caso contrário, **0** é retornado. A função **printList** imprime a lista.

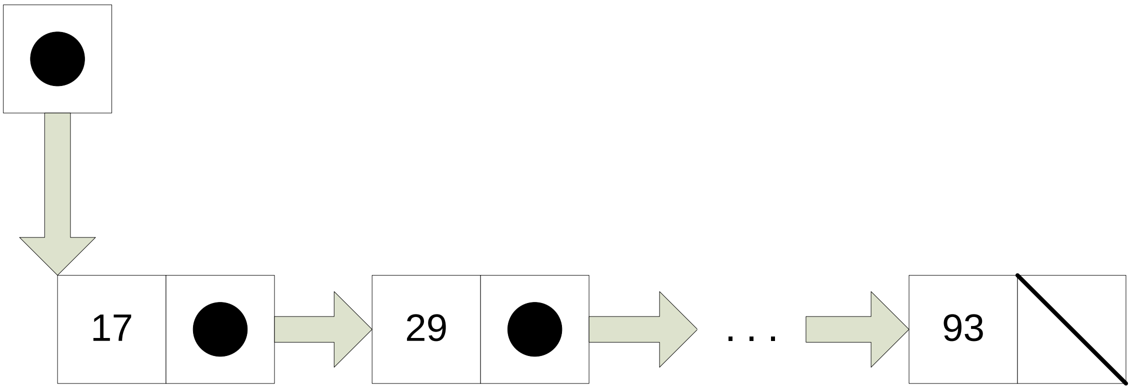


Fig. 12.2 Uma representação gráfica de uma lista encadeada

```
/* Utilizando e mantendo uma lista */
 #include <stdio.h>
#include <stdlib.h>

struct listNode {      /* estrutura auto-referenciada */
            char data;
            struct listNode *hextPtr;
};

typedef struct listNode LISTNODE;
typedef  LISTNODE *LISTNODEPTR;

void insert(LISTNODEPTR *, char);
char delete(LISTNODEPTR *, char);
int isEmpty(LISTNODEPTR);
void printList(LISTNODEPTR);
void instructions(void);

main() {
            LISTNODEPTR startPtr = NULL;
            int choice;
            char item;

            instructions();     /* exibe o menu */
            printf("? ");
            scanf("%d", &choice);

            while  (choice  != 3) {

                        switch (choice) {

                                    case 1:
                                        printf ( "Digite um caractere: ");
                                        scanf("\n%c",  &item);
                                        insert(&startPtr,   item);
                                        printList(startPtr);
                                        break;

                                    case 2:
                                        if  (!isEmpty(startPtr)) {
                                              printf( "Digite o caractere a ser removido: ")
                                              scanf("\n%c",  &item);

                                              if  (delete(SstartPtr,  item)) {
                                                          printf ( "%c removido. \n",  item);
                                                           printList(startPtr);
                                              }
```

```
                                        Else
                                                    printf( "%c nao encontrado.\n\n", item);
                                  }
                                  Else
                                        printf("A lista esta vazia.\n\n");

                                  break;

                              default:
                                    printf( "Escolha invalida.\n\n" );
                                    instructions();
                                    break;
                    }

                    printf("? ");
                    scanf("s&d", &choice);
          }

          printf("Fim do Programa.\n");
          return 0;
}

/* Imprime as instruções */
void instructions(void) {
          printf("Digite sua escolha: \n"
                   "1 para inserir um elemento na lista.\n"
                  "2 para remover um elemento da lista.\n"
                  "3 para finalizar.\n");
}

/* Insere um valor novo na lista, na ordem alfabética */
void insert(LISTNODEPTR *sPtr,  char value){

          LISTNODEPTR newPtr,  previousPtr, currentPtr;

          newPtr = malloc(sizeof(LISTNODE));

          if  (newPtr != NULL)   {        /* existe local disponivel */
                    newPtr->data = value;
                    newPtr->nextPtr  = NULL;

                    previousPtr = NULL;
                    currentPtr = *sPtr;

                    while (currentPtr != NULL && value > currentPtr->data) {
                              previousPtr = currentPtr;  /* vai para ... */
                              currentPtr = currentPtr->nextPtr;     /* ... o no seguinte */
                    }

                    if   (previousPtr == NULL) {
                              newPtr->nextPtr = *sPtr;
```

```
                        *sPtr = newPtr;
                }

                else{
                        previousPtr->nextPtr = newPtr;
                         newPtr->nextPtr = currentPtr;
                }
        }
        Else
          printf( “%c nao inserido. Nao existe memória disponivel.\n" ,  value);

}

/* Remove um elemento da lista */
char delete(LISTNODEPTR *sPtr,  char value)
{
        LISTNODEPTR previousPtr,  currentPtr, tempPtr;

        if  (value ==  (*sPtr)->data) {
                tempPtr = *sPtr;
                *sPtr =  (*sPtr)->nextPtr;      /* retira o no */
                free(tempPtr); /* remove o no retirado */
                return value;
        }

        else{
             previousPtr = *sPtr;
             currentPtr =  (*sPtr)->nextPtr;

             while  (currentPtr  != NULL && currentPtr->data  != value) {

                     previousPtr = currentPtr; /* vai para ... */
                     currentPtr = currentPtr->nextPtr; /* o no seguinte */
             }

             if  (currentPtr  != NULL) {
                     tempPtr = currentPtr;
                     previousPtr->nextPtr = currentPtr->nextPtr;
                     free(tempPtr);
                     return value;
             }
         }
        return  '\0';
}

/* Retorna 1 se a lista estiver vazia, 0 em caso contrario */

int isEmpty(LISTNODEPTR sPtr)
{
        return sPtr == NULL;
)
```

```
/* Imprime a lista */
void printList(LISTNODEPTR currentPtr) {
        if   (currentPtr == NULL)
                printf( "A lista esta vazia.\n\n" ) ;

        else {
                printf( "A lista e:\n");
                while (currentPtr != NULL) {
                        printf("%c  -->  ",  currentPtr->data);
                        evirrentPtr  = currentatr -> nextPtr;
                }

        printf ( "NULL\n\n" ) ;
        }
}
```

Fig. 12.3 Inserindo e removendo nós de uma lista.

```
Digite sua escolha:
1 para inserir um elemento na lista.
2 para remover um elemento da lista.
3 para finalizar.
? 1
Digite um caractere: B A lista e:
B   --> NULL ? 1
Digite um caractere: A
A lista e:
A  -->  B  --> NULL
? 1
Digite um caractere: C A lista e:
A  -->  B  --> C  --> NULL ? 2
Digite o caractere a ser removido: D D nao encontrado.
? 2
Digite o caractere a ser removido: B
B removido.
A lista e:
A   -->  C   --> NULL
? 2
Digite o caractere a ser removido: C C removido. A lista e: A  --> NULL
? 2
Digite o caractere a ser removido: A
A removido.
A lista esta vazia.
? 4
Escolha invalida.
Digite sua escolha:
1 para inserir um elemento na lista.
2 para remover um elemento da lista.
3 para finalizar.
? 3
Fim do programa.
```

Fig. 12.4 Exemplo de saída do programa da Fig. 12.3.

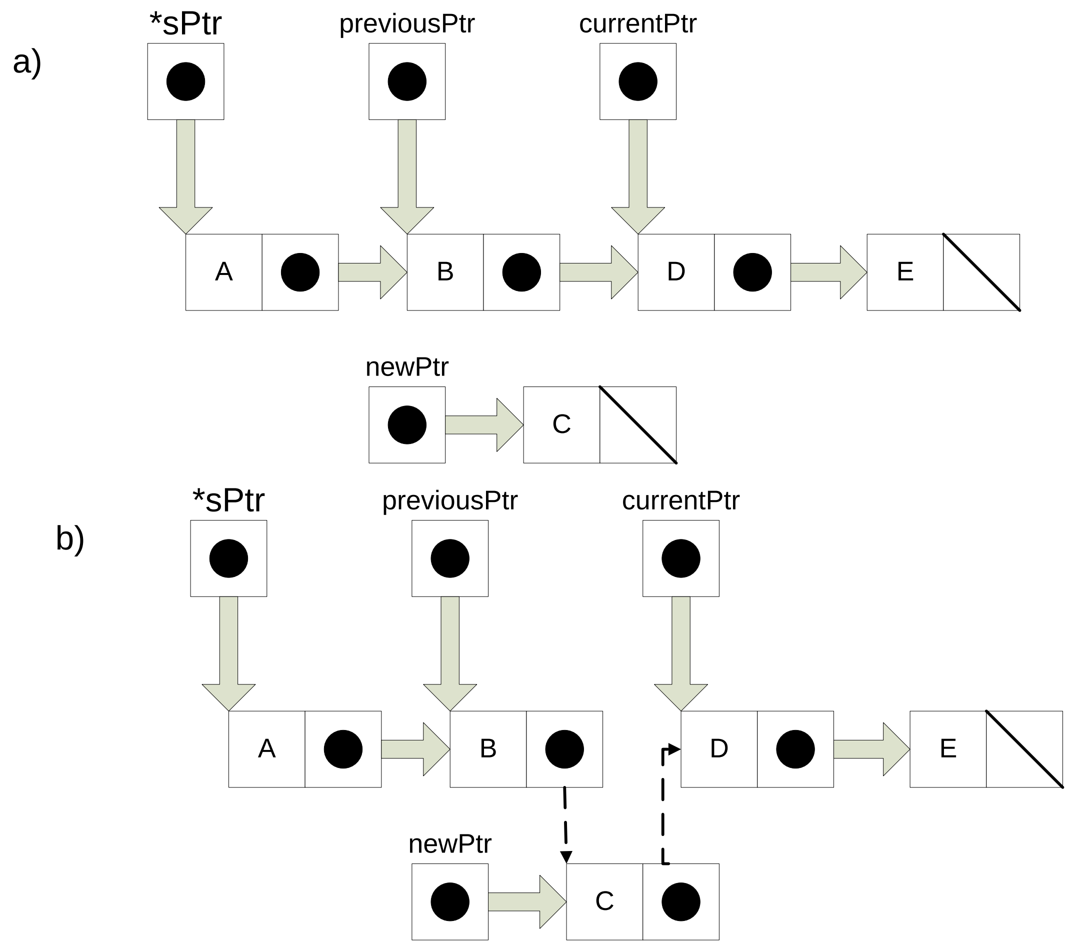


Fig. 12.5 Inserindo um nó em ordem alfabética em uma lista

Os caracteres são inseridos em ordem alfabética na lista. A função **insert** recebe o *endereço* da lista e um caractere a ser inserido. Necessita-se de um endereço da lista quando deve ser inserido um valor no início da lista. Fornecer o endereço da lista permite que ela (i.e., o ponteiro para o prime;: da lista) seja modificada por meio de uma chamada por referência. Como a lista em si é um pontes (para seu primeiro elemento), passar o endereço da lista cria um *ponteiro para um ponteiro* (i.e., *dupla referência indireta).* Isso é um conceito complexo e exige uma programação cuidadosa. As etapas a cumprir para inserir um caractere na lista são as seguintes (veja a Fig. 12.5):

1) Criar um nó chamando **malloc,** atribuindo a **newPtr** o endereço da memória alocada, atribuindo o caractere a ser inserido a **newPtr->data** e atribuindo **NULL** a **newPtr->nextPtr.**

2) Inicializar **previousPtr** com **NULL** e **currentPtr** com ***sPtr** (o ponteiro para o início da lista). Os ponteiros **previousPtr** e **currentPtr** são usados para armazenar os locais do nó situado antes do ponto de inserção e do nó situado após o ponto de inserção.

3) Enquanto **currentPtr** não for **NULL** e o valor a ser inserido for maior do que **currentPtr->data,** atribuir **currentPtr** a **previousPtr** e avançar **currentPtr** para o próximo nó da lista. Isso coloca no lugar adequado na lista o ponto de inserção para o valor considerado.

4) Se **previousPtr** for **NULL,** o novo nó é inserido como primeiro nó na lista. Atribuir ***sPtr** a **newPtr->nextPtr** (o novo nó liga pontos ao primeiro nó anterior) e atribuir **newPtr** a ***sPtr (*sPtr** aponta para o novo nó). Se **previousPtr** não for **NULL,** o novo nó é inserido em seu lugar adequado. Atribuir **newPtr** a **previousPtr->nextPtr** (o nó anterior aponta para o novo nó) e atribuir **currentPtr** a **newPtr->nextPtr** (o link do novo nó aponta para o nó atual).



Boa prática de programação 12.4

*Atribuir **NULL** ao membro de ligação (link) de um novo nó. Os ponteiros devem ser inicializados antes de serem usados.*

A Fig. 12.5 ilustra a inserção de um nó contendo o caractere 'C' na lista ordenada. A parte a) da figura mostra a lista e o novo nó antes da inserção. A parte b) da figura mostra o resultado da inserção do novo nó. Os ponteiros reatribuídos são setas tracejadas.

A função **delete** recebe o endereço do ponteiro no início da lista e um caractere a ser removido.As etapas a cumprir para remover um caractere da lista são as seguintes:

1) Se o caractere a ser removido corresponder ao caractere do primeiro nó da lista, atribuir ***sPtr a tempPtr (tempPtr** será usado para liberar, por meio de **f ree,** a memória desnecessária), atribuir **(*sPtr) ->nextPtr** a ***sPtr** (agora ***sPtr** aponta para o segundo nó na lista), liberar, utilizando **free,** a memória apontada por **tempPtr** e retornar o caractere que foi removido.

2) Caso contrário, inicializar **previousPtr** com ***sPtr** e inicializar **currentPtr** com **(*sPtr)->nextPtr.**

3) Enquanto **currentPtr** não for **NULL** e o valor a ser removido não for igual a **currentPtr->data,** atribuir **currentPtr** a **previousPtr** e atribuir **currentPtr->nextPtr a currentPtr.** Isso localiza o caractere a ser removido se ele estiver contido na lista.

4) Se **currentPtr** não for **NULL,** atribuir **currentPtr** a **tempPtr,** atribuir **currentPtr->nextPtr** a **previousPtr->nextPtr,** liberar o nó apontado por **tempPtr** e retornar o caractere que foi removido da lista. Se **currentPtr** for **NULL,** retornar o caractere **NULL ('\0`)** para indicar que o caractere a ser removido não foi encontrado na lista.

A Fig. 12.6 ilustra a remoção de um nó da lista encadeada. A parte a) da figura mostra a lista encadeada após a operação anterior de inserção. A parte b) mostra a reatribuição do elemento de ligação de **previousPtr** e a atribuição de **currentPtr** a **tempPtr.** O ponteiro **tempPtr** é usado para liberar a memória alocada para armazenar ' **C'.**

A função **printLi st** recebe como argumento um ponteiro para o início da lista e se refere ao **pon**teiro como **currentPtr.** A função determina inicialmente se a lista está vazia. Se estiver, **printList** imprime **"A lista esta vazia"** e encerra o programa. Caso contrário, ela imprime a data na lista. Enquanto **currentPtr** não for **NULL, currentPtr->data** é impresso pela função e **currentPtr->nextPtr** é atribuído a **currentPtr.** Observe que se o link no último nó da lista não for **NULL, o** algoritmo de impressão tentará imprimir além do final da lista e ocorrerá um erro. O algoritmo de impressão é idêntico para listas encadeadas, pilhas e filas.

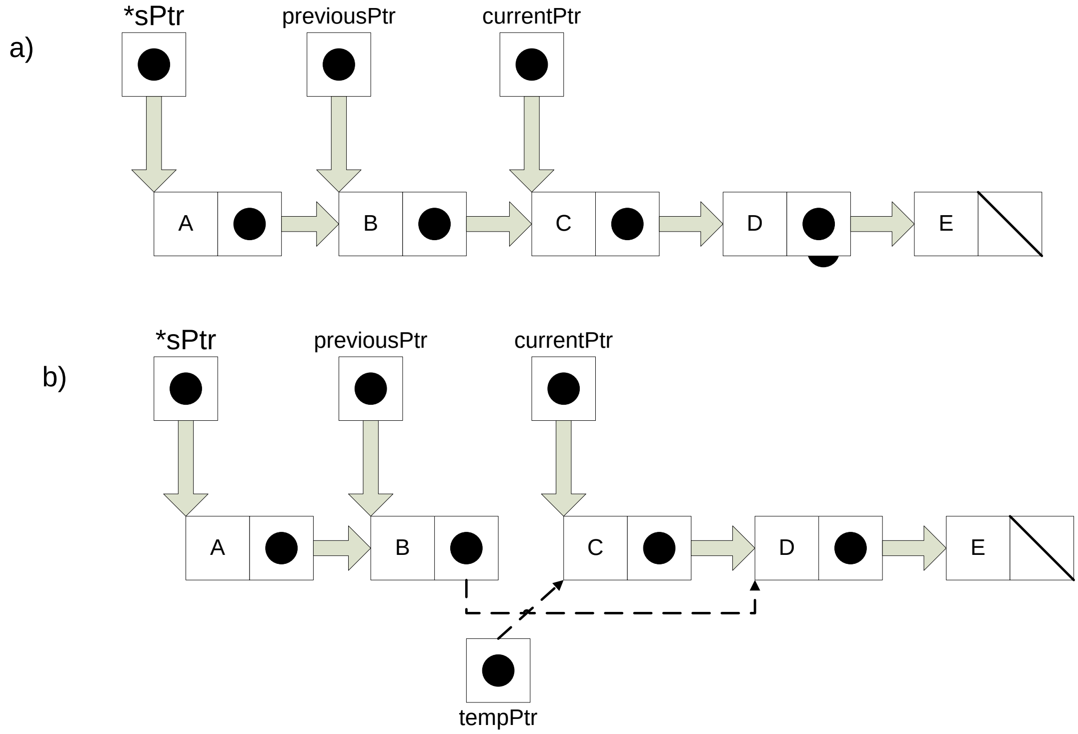


Fig. 12.6 Removendo um nó de uma lista

12.5 Pilhas (Stacks)

Uma *pilha (stack)* é uma versão limitada de uma lista encadeada. Os novos nós só podem ser adicionados no topo da pilha e também só podem ser removidos os nós do topo de uma pilha. Por esse motivo, pilha é conhecida como uma estrutura de dados *último a entrar, primeiro a sair (last-in,first-out,* ou *LIFO)*

A referência a uma pilha é feita por meio de um ponteiro para o elemento do topo da pilha. O membro de ligação (link) no último nó da pilha é definido com **NULL** para indicar a base (o final) da pilha.

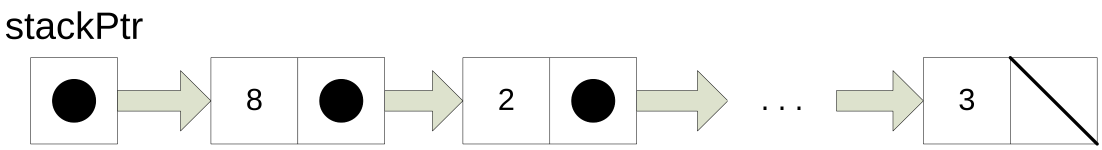
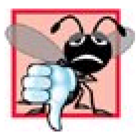


Fig. 12.7 Representação gráfica de uma pilha

A Fig. 12.7 ilustra uma pilha com vários nós. Observe que as pilhas e as listas encadeadas são representadas de maneira idêntica. A diferença entre pilhas e listas encadeadas é que as inserções e remoções podem ocorrer em qualquer lugar de uma lista encadeada, mas apenas no topo de uma pilha.



Erro comun de programação 12.6

*Não definir como **NULL** o link na base de uma pilha.*

As principais funções utilizadas para manipular uma pilha são **push** e **pop.** A função **push** cria um novo nó e o coloca no topo da pilha. A função **pop** remove um nó do tipo de uma pilha, libera a memória que estava alocada ao nó removido e retorna o valor removido.

O programa da Fig. 12.8 (cuja saída está apresentada na Fig. 12.9) implementa uma pilha simples de inteiros. O programa fornece três opções: 1) colocar um valor na pilha (função **push),** 2) remover um valor da pilha (função **pop)** e 3) terminar o programa.

A função **push** coloca um novo nó no topo da pilha. A função consiste em três etapas:

1) Criar um novo nó chamando **malloc,** atribuir o local da memória alocada a **newPtr,** atribuir a **newPtr->data** o valor a ser colocado na pilha e atribuir **NULL** a **newPtr->nextPtr.**

2) Atribuir * **topPtr** (o ponteiro do topo da pilha) a **newPtr- >nextPtr** — o membro de ligação de **newPtr** aponta agora para o nó que estava anteriormente no topo.

3) Atribuir **newPtr** a ***topPtr** — agora ***topPtr** aponta para o novo topo da pilha.

As manipulações envolvendo ***topPtr** mudam o valor de **stackPtr emmain.** A Fig. 12.10 ilustra a função **push.** A parte a) da figura mostra a pilha e o novo nó antes de **push** ser executada. As linhas tracejadas na parte b) ilustram as etapas 2 e 3 da execução de **push** que permitem ao nó que contém o **valor 12** se tornar o novo topo da pilha.

A função **pop** remove um nó do topo da pilha. Observe que **main** determina, antes de chamar **pop,** se a pilha está vazia. A execução de **pop** consiste em cinco etapas.

1) Atribuir ***topPtr** a **tempPtr (tempPtr** será usado para liberar a memória desnecessária).

2) Atribuir **(*topPtr) ->data** a **popValue** (salva o valor armazenado no nó de topo).

3) Atribuir **(*topPtr) ->nextPtr** a ***topPtr** (atribui a ***topPtr** o endereço do novo nóde topo).

4) Liberar a memória apontada por **tempPtr.**

5) Retornar **popValue** à função chamadora **(main** no programa da Fig. 12.8).

A Fig. 12.11 ilustra a função **pop.** Aparte a) mostra a pilha depois da execução anterior de **push.** A parte b) mostra **tempPtr** apontando para o primeiro nó da pilha e **topPtr** apontando para o segundo nó da pilha. A função **free** é usada para liberar a memória apontada por **tempPtr.**

As pilhas possuem muitas aplicações interessantes. Por exemplo, sempre que for feita a chamada de uma função, a função chamada deve saber como retornar à função chamadora, portanto o endereço de retorno é colocado em uma pilha. Se ocorrer a chamada de uma série de funções, os sucessivos valores de retorno são colocados na pilha, na ordem último a entrar, primeiro a sair, de forma que cada função possa retornar a quem a chamou. As pilhas suportam chamadas recursivas de funções da mesma forma que chamadas convencionais não-recursivas.

As pilhas contêm o espaço criado para variáveis automáticas em cada chamada de uma função. Quando a função retorna a quem a chamou, o espaço daquelas variáveis automáticas é removido da pilha e as variáveis deixam de ser conhecidas pelo programa.

```
/* programa de pilha dinâmica */
#include <stdio.h>
 #include <stdlib.h>

struct stackNode {    /* estrutura auto-referenciada */
        int data;
        struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef  STACKNODE *STACKNODEPTR;

void push(STACKNODEPTR *, int);
int pop(STACKNODEPTR *);
int  isEmpty(STACKNODEPTR);
```

```
void printStack(STACKNODEPTR);
void instructions(void);

main() {

      STACKNODEPTR stackPtr = NULL; /* pontos para o topo da pilha */
      int choice, value;
      instructions();
      printf("?  ");
      scanf("%d", &choice);
      while  (choice  != 3) {
         switch  (choice) {
               case 1: /* coloca valor na pilha */
                        printf("Digite um inteiro:   ");
                        scanf("%d",  &value);
                        push(&stackPtr, value);
                        printStack(stackPtr);
                        break;
               case 2 : /* retira valor da pilha */
                        if  (!isEmpty(stackPtr))
                           printf("O valor retirado e %d.\n", pop(&stackPtr));
                           printStack(stackPtr) ;
                           break;
               default:
                        printf( "Escolha invalida.\n\n" );
                        instructions();
                        break;
         }
         printf("? ");
         scanf("%d",   &choice);
      }
      printf("Fim do programa.\n");
      return 0;
}

/* Imprime as instruções */
void instructions(void)
{
         printf( "Digite uma escolha:\n"
                 "1 para colocar um valor na pilha\n"
                 "2 para retirar um valor da pilha\n"
                 "3 para finalizar o programa\n");
}

/* Insere um valor no topo da pilha */
void push(STACKNODEPTR *topPtr,   int info){

    STACKNODEPTR newPtr;
    newPtr = malloc(sizeof(STACKNODE));
    if  (newPtr  != NULL) {
       newPtr->data = info;
```

```
        newPtr->nextPtr = *topPtr;
        *topPtr = newPtr;
    }
    else
        printf( "%d nao foi inserido. Nao existe memória disponivel.\n" ,  info);
}

/* Remove um valor do topo da pilha */
int  pop(STACKNODEPTR *topPtr){
        STACKNODEPTR tempPtr;
        int popValue;
        tempPtr = *topPtr;
        popValue =  (*topPtr)->data;
        *topPtr =   (*topPtr)->nextPtr;
        free(tempPtr);
        return popValue;
}

/* Imprime a pilha */
void printStack(STACKNODEPTR currentPtr){
        if  (currentPtr == NULL)
                printf( "A pilha esta vazia.\n\n" );
        else {
                printf("A pilha e:\n");
                while   (currentPtr   != NULL) {
                        printf ('"%d --> ",  currentPtr->data) ;
                        currentPtr = currentPtr->nextPtr;
                }
        printf("NULL\n\n");
}
/* A pilha esta vazia? */
int isEmpty(STACKNODEPTR topPtr) {
        return topPtr == NULL;
}
```

Fig. 12.8 Um programa simples de pilha.

```
Digite uma escolha:
1 para colocar um valor na pilha
2 para retirar um valor da pilha
3 para finalizar o programa ? 1
Digite um inteiro: 5 A pilha e:
5 --> NULL
? 1
Digite um inteiro: 6 A pilha e:
6 -->  5  --> NULL
? 1
Digite um inteiro: 4 A pilha e:
4 -->  6  -->  5  --> NULL ? 2
O valor retirado e 4.
A pilha e:
```

```
6   -->   5   --> NULL
? 2
O valor retirado e 6. A pilha e:
5 --> NULL
? 2
0 valor retirado e 5. A pilha esta vazia.
? 2
A pilha esta vazia. ? 4
Escolha invalida.
Digite uma escolha:
1 para colocar um valor na pilha
2 para retirar um valor da pilha
3 para finalizar o programa
? 3
Fim do programa.
```

Fig. 12.9 Exemplo de saída do programa da Fig. 12.8.

As pilhas são usadas por compiladores no processo de cálculo de expressões e na geração do código de linguagem de máquina. Os exercícios exploram várias aplicações das pilhas.

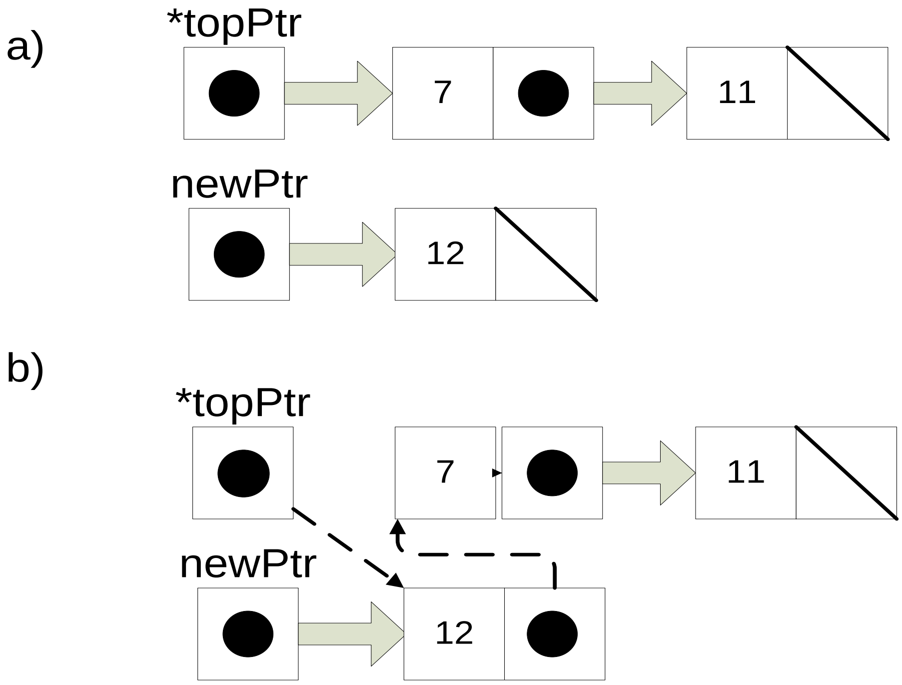


Fig. 12.10 A execução de **push.**

12.6 Filas (Queues)

Outra estrutura comum de dados é a *fila (queue).* Uma fila funciona como um atendimento em uma mercearia — a primeira pessoa da fila é atendida em primeiro lugar e os outros clientes entram na fila somente no final e esperam ser servidos. Os nós são removidos apenas do *início* das filas e são inseridos apenas em seu *final.* Por esse motivo, uma fila é conhecida como uma estrutura de dados do tipo *primeiro a entrar, primeiro a sair (first-in, first-out* ou *FIFO).* As operações de inserção e remoção são conhecidas como *enfileirar (enqueue)* e *desenfileirar (ãequeue).*

As filas possuem muitas aplicações em sistemas computacionais. Muitos computadores possuem apenas um único processador, portanto apenas um usuário pode ser servido de cada vez. As entradas de outros usuários são colocadas em uma fila. Cada entrada avança gradualmente para a frente da fila à medida que outros usuários são servidos. A entrada na frente da fila é a próxima a ser servida.

As filas também são usadas para suportar armazenamento de dados (spooling) para impressão. Um ambiente multiusuário pode ter apenas uma única impressora. Muitos usuários podem estar gerando saídas impressas. Se a impressora estiver ocupada, outras saídas ainda podem estar sendo geradas. Estas são "armazenadas" ("spooled") em disco onde esperam em uma fila até que a impressora fique disponível.

Os pacotes de informações também esperam em filas em redes de computadores. Cada vez que um pacote chega a um nó da rede, tal pacote deve ser direcionado para o próximo nó da rede situado ao longo do trajeto do pacote até seu destino final. O nó de roteamento dirige um pacote por vez, portanto pacotes adicionais são enfileirados até que o roteador possa enviá-los. A Fig. 12.12 ilustra uma fila com vários nós. Observe os ponteiros para o início (head) e o final (tail) da fila.

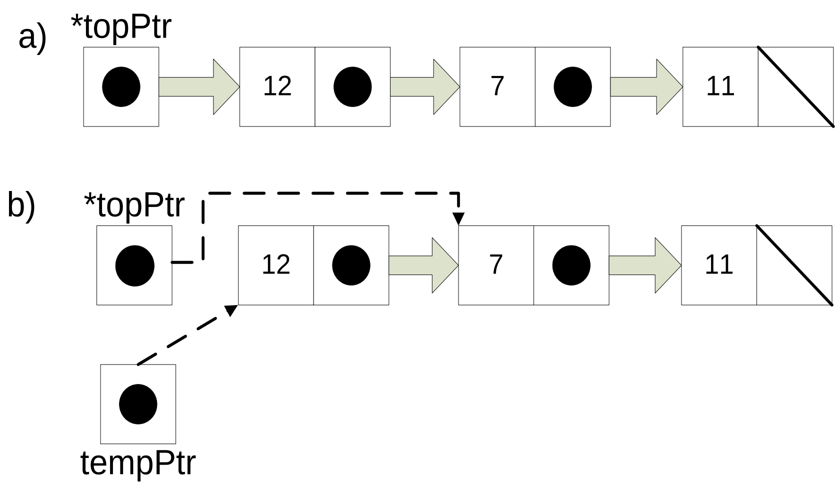


Fig. 12.11 A execução de **pop.**

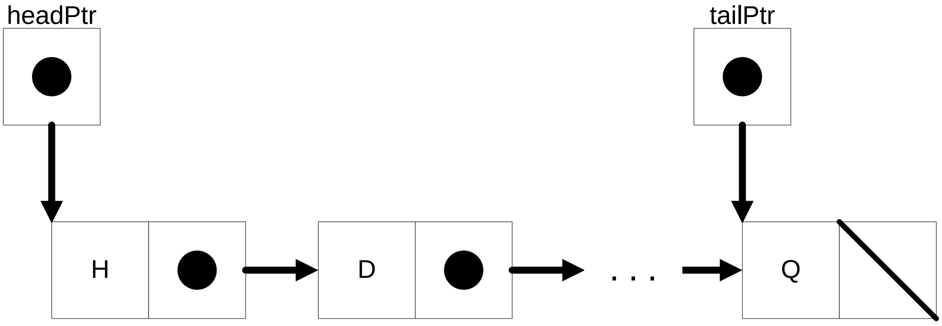


Fig. 12.12 Uma representação gráfica de uma fila

Erro comun de programação 12.7

*Não definir como **NULL** o link no último nó de uma fila.*

O programa da Fig. 12.13 (cuja saída está na Fig. 12.14) realiza manipulações de filas. O programa fornece várias opções: inserir um nó na fila (função **enqueue),** remover um nó de uma fila (função **dequeue)** e terminar o programa.

```
/* Utilizando e mantendo uma fila */
#include <stdio.h>
 #include <stdlib.h>

struct queueNode  { /* estrutura auto-referenciada */
            char data;
            struct queueNode *nextPtr;
};

typedef struct queueNode QUEUENODE;
typedef  QUEUENODE  *QUEUENODEPTR;

/* protótipos de funções */

void printQueue(QUEUENODEPTR);
int  isEmpty(QUEUENODEPTR);
char dequeue(QUEUENODEPTR  *,   QUEUENODEPTR *);
void enqueue(QUEUENODEPTR *,   QUEUENODEPTR *,   char);
void instructions(void);


main() {

          QUEUENODEPTR headPtr  =  NULL,   tailPtr  = NULL;
          int choice;
          char item;

          instructions();
          printf("? ");
```

```
        scanf("%d",   &choice);
            while  (choice  != 3) {
                switch(choice) {
                    case 1:
                        printf( "Digite um caractere: ");
                        scanf("\n%c",  &item);
                        enqueue(&headPtr,  stailPtr,   item);
                        printQueue(headPtr);
                        break;
                    case 2:
                        if  (!isEmpty(headPtr)) {
                            item = dequeue(&headPtr, &tailPtr);
                            printf("%c foi desenfileirado.\n",   item);
                            printQueue(headPtr);
                            break;
                    default:

                        printf("Escolha invalida.\n\n");
                        instructions();
                        break;
                }

            printf ("? ");
            scanf("%d",  fcchoice);
            }

        printf( "Fim do programa.\n" );
        return 0;
}

void instructions(void){
        printf ("Digite sua escolha:\n"
                "1 para adicionar um item a fila.\n"
                “2 para remover um item da fila.\n"
                “3 para finalizar.\n"
}

void enqueue(QUEUENODEPTR *headPtr,  QUEUENODEPTR *tailPtr, char value){

        QUEUENODEPTR newPtr;
        newPtr = malloc(sizeof(QUEUENODE));

        if (newPtr != NULL) {
                newPtr->data = value;
                newPtr->nextPtr = NULL;
                if  (isEmpty(*headPtr))
                        *headPtr = newPtr;

                Else
                        (*tailPtr)->nextPtr = newPtr; *tailPtr = newPtr;
        }
```

```
        Else
                printf( "%c nao inserido. Nao existe memória disponivel.\n", value);
}

char dequeue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr) {
        char value;
        QUEUENODEPTR tempPtr;
        value =   (*headPtr)->data;
        tempPtr = *headPtr;
        *headPtr =  (*headPtr)->nextPtr;
        if  (*headPtr == NULL)
                tailPtr = NULL;
        free(tempPtr);
        return value;
}
int  isEmpty(QUEUENODEPTR headPtr) {
        return headPtr == NULL;
}

void printQueue(QUEUENODEPTR currentPtr) {

        if  (currentPtr == NULL)
                printf("A fila esta vazia.\n\n" ) ;
        else {
                printf ( "A fila e:  \n" );

                while  (currentPtr  != NULL) {
                        printf("%c --> ",  currentPtr->data);
                        currentPtr = currentPtr->nextPtr;
                }
                printf("NULL\n\n");
        }
}
```

Fig. 12.13 Processando uma fila.

A função **enqueue** recebe três argumentos de **main:** o endereço do ponteiro do início da fila, o endereço do ponteiro do final da fila e o valor a ser inserido na fila. A função é executada em três etapas:

1) Para criar um novo nó: Chamar **malloc,** atribuir o local da memória alocada a **newPtr,** atribuir a **newPtr->data m**o valor a ser inserido na fila e atribuir **NULL** a **newPtr->nextPtr.**

2) Se a fila estiver vazia, atribuir **newPtr** a ***headPtr;** caso contrário, atribuir o ponteiro **newPtr** a **(*tailPtr)->nextPtr.**

3) Atribuir **newPtr** a ***tailPtr.**

A Fig. 12.15 ilustra o funcionamento de **enqueue.** A parte a) da figura mostra a fila e o novo nó antes da função ser executadas. As linhas tracejadas da parte b) ilustram as etapas 2 e 3 da função **enqueue** que permitem que um novo nó seja adicionado ao final de uma fila que não esteja vazia.

A função dequeue recebe como argumentos o endereço do ponteiro para o início da fila e o endereço do ponteiro para o final da fila e remove o primeiro nó da fila. A execução de dequeue consiste em seis etapas:

1) Atribuir **(*headPtr) ->data** a **value** (salvar os dados).

2) Atribuir ***headPtr** a **tempPtr (tempPtr** é usado para liberar, por meio de **free,** a memória.

3) Atribuir **(*headPtr) ->nextPtr** a ***headPtr (*headPtr** aponta agora para o novo primeiro nó na fila).

4) Se ***headPtr** for **NULL,** atribuir **NULL** a ***tailPtr.**

5) Liberar a memória apontada por **tempPtr.**

6) Retornar **value** à função chamadora (a função **dequeue** é chamada por **main** no programa da Fig. 12.13).

```
Digite sua escolha:
1 para adicionar um item a fila
2 para remover um item da fila
3 para finalizar
? 1
Digite um caractere: A A fila e:
A  --> NULL 4 ? 1
Digite um caractere: B
A fila e:
A -->  B --> NULL
? 1
Digite um caractere: C A fila e:
A -->  B --> C --> NULL ? 2
A foi desenfileirado.
A fila e:
B   -->  C  --> NULL
? 2
B foi desenfileirado. A fila e:
C   --> NULL ? 2
C foi desenfileirado. A fila esta vazia.
? 2
A fila esta vazia.
? 4
Escolha invalida.
Digite sua escolha:
1 para adicionar um item a fila
2 para remover um item da fila
3 para finalizar
? 3
Fim do programa.
```

Fig. 12.14 Exemplo de saída do programa da Fig. 12.13.

A Fig. 12.16 ilustra a função **dequeue.** A parte a) mostra a fila após a execução anterior de **enqueue.** A parte b) mostra **tempPtr** apontando para o nó desenfileirado e **headPtr** apontando para o novo primeiro nó da fila. A função **free** é usada para recuperar a memória apontada por **tempPtr.**

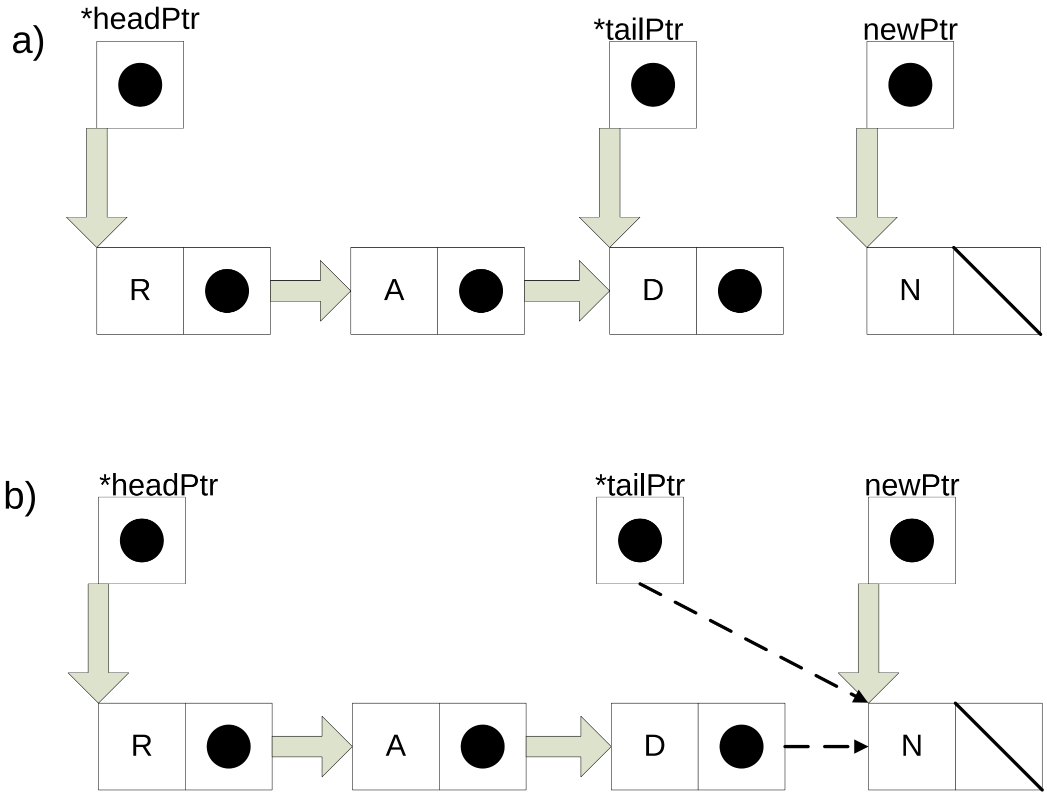


Fig. 12.15 Uma representação gráfica da execução de **enqueue**.

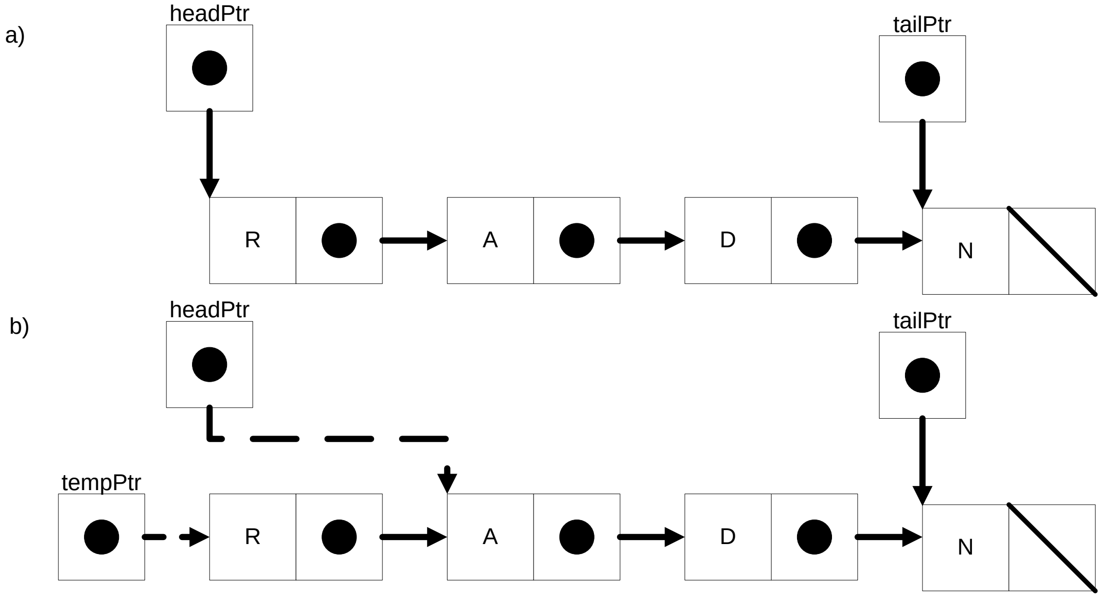


Fig. 12.16 Uma representação da execução de **dequeue**.

12.7 Arvores

As listas encadeadas, pilhas e filas são *estruturas lineares de dados.* Uma árvore é uma estrutura de dados não-linear e bidimensional com propriedades especiais. Os nós da árvore contêm dois ou mais links. Esta seção analisa as *árvores binárias* (Fig. 12.17) — árvores cujos nós contêm dois links (nenhum, um ou ambos dos quais podem ser **NULL).** O *nó raiz {nó principal* ou *root node)* é o primeiro nó da árvore. Cada link do nó raiz se refere a um *filho (child). O filho da esquerda (left child)* é o primeiro nó na *subárvore esquerda (left subtree)* e o *filho da direita (right child)* é o primeiro nó na *subárvore direita (right subtree).* Os filhos de um nó são chamados *irmãos (siblings).* Um nó sem filhos é chamado um *nó folha.* Normalmente, os cientistas computacionais desenham árvores do nó raiz para baixo — de forma exatamente oposta à das árvores na natureza.

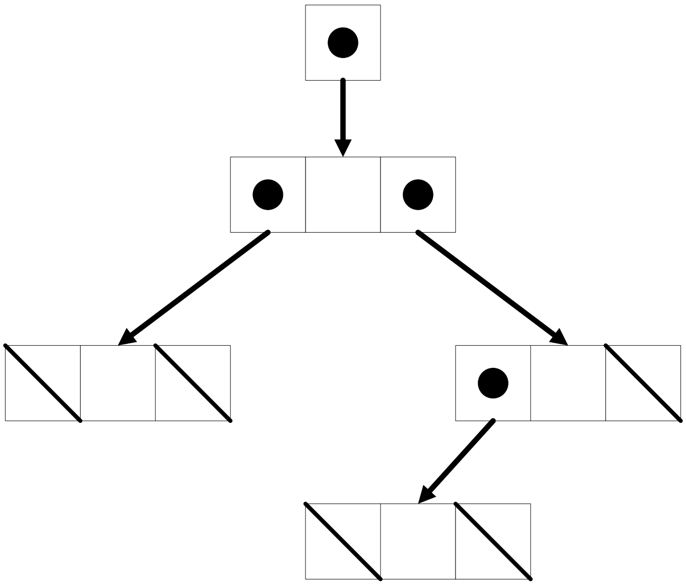
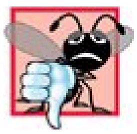


Fig. 12.16 Uma representação gráfica de uma arvore binária.

Nesta seção, é criada uma árvore binaria especial chamada *árvore de pesquisa binaria.* Uma árvore de pesquisa binaria (sem valores duplicados de nós) apresenta a característica de que os valores em qualquer subárvore esquerda são menores do que o valor de seu nó pai e os valores de qualquer subárvore direita são maiores do que o valor em seu nó pai. A Fig. 12.18 ilustra uma árvore de pesquisa binária com 12 valores. Observe que o formato de uma árvore binaria que corresponde a um conjunto de dados pode variar em função da ordem na qual os valores são inseridos na árvore.



Erro comun de programação 12.8

Não definir como ***NULL*** *os links nos nós folhas de uma árvore.*

O programa da Fig. 12.19 (cuja saída está mostrada na Fig. 12.20) cria uma árvore de pesquisa binária e a percorre de três maneiras — *inorder* (ou *in-ordem), preorder* (ou *pré-ordem)* e *postorder* (ou *pós-ordem).* O programa gera 10 números aleatórios e os insere na árvore, descartando os valores duplicados.

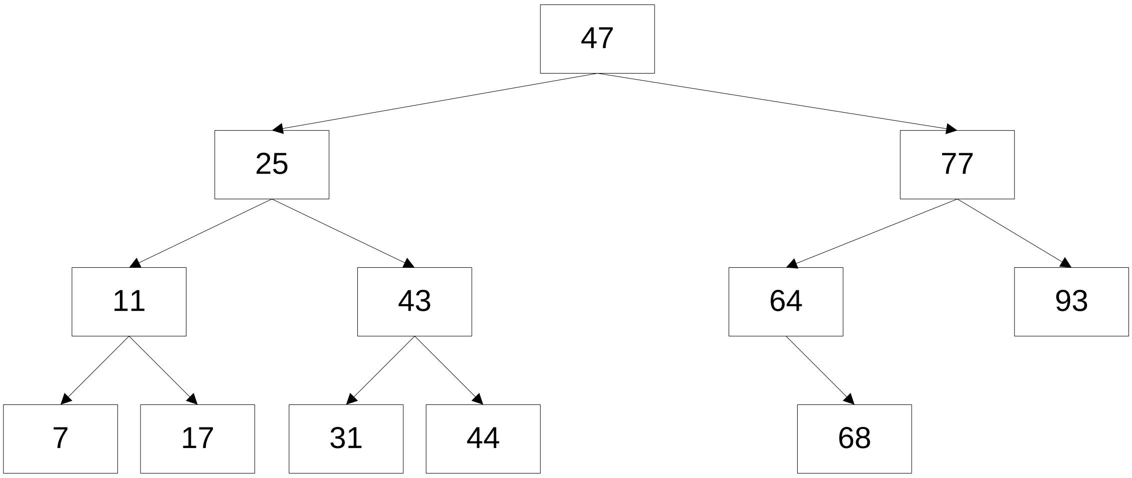


Fig. 12.18 Uma árvore de pesquisa binária.

As funções usadas na Fig. 12.19 para criar uma árvore de pesquisa binaria e percorrê-la são recursivas. A função **insertNode** recebe como argumentos o endereço da árvore e um inteiro para ser armazenado na árvore. *Um nó só pode ser inserido na árvore de pesquisa binaria como um nó folha.* As etapas para inserir um nó em uma árvore de pesquisa binaria são as seguintes:

1) Se ***treePtr** for **NULL,** criar um novo nó. Chamar **malloc,** atribuir a memória alocada a ***treePtr,** atribuirá **(*treePtr) ->data** o inteiro a ser armazenado, atribuirá **(*treePtr) ->leftPtr** e a **(*treePtr) ->rightPtr** o valor **NULL** e retornar à função chamadora (seja ela **main** ou uma chamada anterior a **insertNode).**

2) Se o valor de ***treePtr** não for **NULL** e o valor a ser inserido for menor do que^ ***treePtr) ->data,** a função **insertNode** é chamada com o endereço de **(*treePtr) ->lef tPtr.** Caso contrário, a função **insertNode** é chamada com o endereço de **(*treePtr) ->rightPtr.** As etapas recursivas continuam até que um ponteiro **NULL** seja encontrado, quando então a etapa 1) é executada para inserir um novo nó.

```
/* Cria uma arvore binaria e a percorre em preorder, inorder e postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
            struct treeNode *leftPtr;
             int data;
            struct treeNode *rightPtr;
};

typedef struct treeNode TREENODE;
typedef TREENODE *TREENODEPTR;

void insertNode(TREENODEPTR *,   int);
void inOrder(TREENODEPTR) ;
void preOrder(TREENODEPTR);
void postorder(TREENODEPTR);

```

```
main(){

     int i, item;
     TREENODEPTR rootPtr  = NULL;
     srand(time(NULL)); /* tenta inserir 10 valores aleatórios entre 0 e 14 na arvore */
     printf( "Os números que estão sendo colocados na arvore sao:\n"  ) ;
     for  (1=1;  i <= 10;  i++) {
            item = rand() % 15;
            printf("%3d",  item);
            insertNode(fcrootPtr, item);
     }
     /* percorre a arvore com preOrder */
     printf(" \n\nO percurso com preOrder e:\n");
     preOrder(rootPtr);

     /* percorre a arvore com inOrder */
     printf( "\n\nO percurso com inOrder e:\n" );
     inOrder(rootPtr);

     /* percorre a arvore com postorder */
     printf("\n\nO percurso com postorder e:\n");
     postOrder(rootPtr);
     return 0;
}

void insertNode(TREENODEPTR *treePtr,   int value){

     if  (*treePtr == NULL)   {    /* *treePtr tem valor NULL */
          *treePtr = malloc(sizeof(TREENODE));

     if  (*treePtr  != NULL) {
          (*treePtr)->data = value;
          (*treePtr)->leftPtr = NULL;
          (*treePtr)->rightPtr = NULL;
     }
     else{
          printf ( "%d nao inserido. Nao existe memória disponivel. \n",value);
     }
     else{
          if  (value <   (*treePtr)->data)
               insertNode(&((*treePtr)->leftPtr),  value);
          else
               if  (value >   (*treePtr)->data)
                    insertNode(&((*treePtr)->rightPtr),  value);
               else
                    printf("dup");
}

void inOrder(TREENODEPTR treePtr) {
     if   (treePtr  != NULL) {
          inOrder(treePtr->leftPtr) ;
```

```
            printf("%3d",  treePtr->data);
            inOrder(treePtr->rightPtr) ;
        }
}

void preOrder(TREENODEPTR treePtr)
{
        if  (treePtr  != NULL) {
            printf("%3d",  treePtr->data);
            preOrder(treePtr->leftPtr);
            preOrder(treePtr->rightPtr);
        }
}

void postOrder(TREENODEPTR treePtr) {
        if  (treePtr  != NULL) {
            postOrder(treePtr->leftPtr);
            postOrder(treePtr->rightPtr);
            printf("%3d",   treePtr->data);
        }
}
```

Fig. 12.19 Criando e percorrendo uma árvore binária**.**

```
Os números que estão sendo colocados na arvore sao:
 7    8    0    6 14    1    Odup 13    Odup 7dup

O percurso com preOrder e:
7    0    6    1    8    14    13

O percurso com inOrder e:
0    1    6    7    8    13    14

O percurso com postorder e:
1    6    0    13  14   8      7
```

Fig. 12.20 Exemplo de saída do programa da Fig. 12.19.

Cada uma das funções **inOrder, preOrder** e **postorder** recebe uma árvore (i.e., o ponteiro para o nó raiz da árvore) e a percorre.

Os passos para um percurso (travessia) **inOrder** são:

1) Percorrer em **inOrder** a subárvore esquerda.

2) Processar o valor no nó.

3) Percorrer em **inOrder** a subárvore direita.

O valor em um nó não é processado até que os valores em sua subárvore esquerda sejam processados. O percurso **inOrder** da árvore da Fig. 12.21 é:

6 13 17 27 33 42 48

Observe que o percurso **inOrder** de uma árvore de pesquisa binaria imprime os valores dos nós **na** ordem ascendente. Na realidade, o processo de criar uma árvore de pesquisa binaria classifica os dados — e, por isso, esse processo é chamado *classificação de árvore binária.*

Os passos para um percurso (travessia) **preOrder** são:

1) Processar o valor no nó.
2) Percorrer em **preOrder** a subárvore esquerda.
3) Percorrer em **preOrder** a subárvore direita.

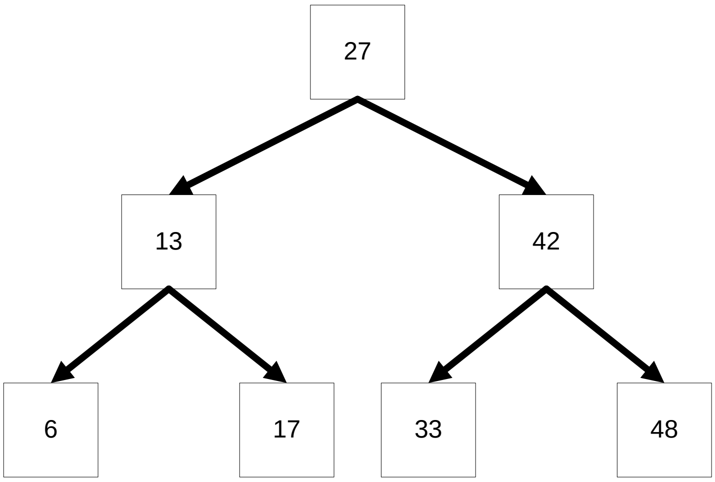


Fig. 12.21 Uma árvore de pesquisa binária.

O valor em cada nó é processado à medida que o nó é visitado. Depois de o valor de um determinado nó ser processado, são processados os valores da subárvore esquerda e então os valores da subárvore direita. O percurso **preOrder** da árvore da Fig. 12.21 é:

27 13 6 17 42 33 48

Os passos para um percurso (travessia) **postOrder** são:

1) Percorrer em **postOrder** a subárvore esquerda.

2) Percorrer em **postOrder** a subárvore direita.

3) Processar o valor no nó.

O valor em cada nó não é impresso até que os valores em seus filhos sejam impressos. O percurso **postOrder** da árvore da Fig. 12.21 é:

6 17 13 33 48 42 27

A árvore de pesquisa binaria facilita a eliminação de valores duplicados. À medida que a árvore é criada, uma tentativa de inserir um valor duplicado será

reconhecida porque a duplicata enfrentará as mesmas decisões que o valor original de "ir para a esquerda" ou "ir para a direita" em cada comparação. Dessa forma, posteriormente a duplicata será comparada com um nó que contém o mesmo valor. Nesse momento a duplicata pode ser simplesmente descartada.

Procurar em uma árvore binaria um valor que corresponde a um valor-chave também é rápido. Se a árvore estiver montada corretamente, cada nível contém cerca de duas vezes o número de elementos do nível anterior. Portanto uma busca binaria com *n* elementos teria um máximo de $\log_2$« níveis e assim precisariam ser feitas no máximo $\log_2 n$ comparações para encontrar uma correspondência ou para determinar que ela não existe. Isso significa, por exemplo, que comparar uma árvore binaria de 1000 elementos (bem compactada) não exigiria mais de 10 comparações porque $2^{10} > 1000$. Para pesquisar uma árvore de pesquisa binaria (bem compactada) com 1.000.000 elementos não seriam necessárias mais de 20 comparações porque $2^{20} > 1.000.000$.

Nos Exercícios, são apresentados algoritmos para várias outras operações de árvores binárias como remoção (eliminação) de um item de uma árvore binaria, imprimir uma árvore binaria em um formato de árvore bidimensional e realizar uma travessia (percurso) segundo a ordem dos níveis de uma árvore binaria. O percurso segundo a ordem dos níveis de uma árvore binaria visita os nós da árvore, linha após linha, começando no nível do nó raiz. Em cada nível da árvore, os nós são visitados da esquerda para a direita. Outros exercícios de árvores binárias incluem permitir a existência de valores duplicados em uma árvore binaria, inserir valores strings em uma árvore binaria e determinar quantos níveis estão contidos em uma árvore binaria.

Resumo

• As estruturas auto-referenciadas contêm membros chamados links (ligações ou encadeamentos) que apontam para estruturas do mesmo tipo.

• As estruturas auto-referenciadas permitem que muitas estruturas sejam ligadas entre si em pilhas, filas, listas e árvores.

• A alocação dinâmica da memória reserva um bloco de bytes na memória para armazenar os objetos de dados durante a execução de um programa.

• A função **malloc** utiliza como argumento o número de bytes a serem alocados e retorna um ponteiro **void** para a memória alocada. A função **malloc** é usada normalmente com o operador **sizeof.** O operador **sizeof** determina o tamanho em bytes da estrutura para a memória que está sendo alocada.

• A função **free** libera memória.

• Uma lista encadeada (ou lista linear) é um conjunto de dados armazenados em um grupo de estruturas auto-referenciadas conectadas entre si.

• Uma lista encadeada é uma estrutura dinâmica de dados — o comprimento da lista pode aumentar ou diminuir quando necessário.

• As listas encadeadas podem continuar a aumentar enquanto houver memória disponível.

• As listas encadeadas fornecem um mecanismo para inserção e remoção simples de dados por meio da re-atribuição de ponteiros.

• As pilhas (stacks) e filas (queues) são versões especializadas de uma lista encadeada.

• Apenas no topo de uma pilha são adicionados novos nós e removidos nós existentes. Por esse mostivo, uma pilha é conhecida como uma estrutura de dados do tipo último a entrar, primeiro a sair (last in, first-out, ou LIFO).

• O membro de ligação do último nó de uma pilha é definido como **NULL** para indicar a base (o final) da pilha.

• As duas operações principais para manipular uma pilha são push e pop. A operação push cria um novo nó e o coloca no topo da pilha. A operação pop remove um nó do topo de uma pilha, libera a memória que estava alocada ao nó removido e retorna o valor removido.

• Em estrutura de dados em fila, os nós são removidos no topo e adicionados ao final. Por esse motivo, uma fila é conhecida como uma estrutura de dados do tipo primeiro a entrar, primeiro a sair (first-in, first-out, ou FIFO). As operações de adicionar e remover são conhecidas como enfileirar (enqueue) e desenfileirar (dequeue).

• As árvores são estruturas de dados mais complexas do que as listas encadeadas, filas e pilhas. As árvores são estruturas bidimensionais de dados que exigem dois ou mais links por nó.

• As árvores binárias contêm dois links por nó.

• O nó raiz é o primeiro nó da árvore.

• Cada um dos ponteiros do nó raiz se refere a um filho. O filho da esquerda é o primeiro nó na subárvore da esquerda e o filho da direita é o primeiro nó da subárvore da direita. Os filhos de um nó são chamados irmãos (siblings). Se um nó não tiver filhos é chamado nó folha (leaf node).

• Uma árvore de pesquisa binaria tem a característica que o valor do filho da esquerda de um nó é menor que o valor do nó pai, e o valor do filho da direita de um nó é maior ou igual ao valor do nó pai. Se puder ser determinado que não há valores duplicados de dados, o valor do filho da direita é simplesmente maior que valor do nó pai.

• Um percurso (travessia) inorder (in-ordem) de uma árvore binaria percorre em inorder a subárvore da esquerda, processa o valor do nó e percorre em inorder a subárvore da direita. O valor de um nó não é processado até que os valores na subárvore da direita sejam processados.

• Um percurso preorder (pré-ordem) processa o valor no nó, percorre em preorder a subárvore da esquerda e percorre em preorder a subárvore da direita. O valor em cada nó é processado à medida que o nó é encontrado.

• Um percurso postorder (pós-ordem) percorre em postorder a subárvore da esquerda, percorre em postorder a subárvore da direita e processa o valor do nó. O valor em cada nó não é processado até que os valores em ambas as subárvores sejam processados.

Terminologia

alocação dinâmica de memória
árvore
árvore binaria
árvore de pesquisa binaria
classificação de árvore binaria
dequeue
desenfileirar
dupla referência indireta
eliminando um nó
enfileirar
enqueue
estrutura auto-referenciada
estrutura linear de dados
estrutura não-linear de dados
estruturas dinâmicas de dados
FIFO (first-in, first-out)
fila
filho da direita
percurso inorder (in-ordem)
percurso postorder (pós-ordem)
percurso preorder (pré-ordem)
pilha
ponteiro **NULL**
ponteiro para um ponteiro
pop
primeiro a entrar, primeiro a sair
push
queue
removendo um nó

siblings
sizeof
filho da esquerda
filhos
final de uma fila
free
função predicada início de uma fila inserir um nó irmãos
LIFO (last-in, first-out)
lista encadeada
malloc (alocar memória)
nó
nó filho
nó filha
nó pai
nó principal
nó raiz percurso
Stack
subárvore
subárvore direita
subárvore esquerda
subtree
topo
travessia
travessia inorder (in-ordem)
travessia postorder (pós-ordem)
travessia preorder (pré-ordem)
tree
último a entrar, primeiro a sair
visitar um nó

Erros Comuns de Programação

12.1 Não definir como **NULL** o link no último nó de uma lista.

12.2 Admitir que o tamanho de uma estrutura é simplesmente a soma dos tamanhos de seus membros.

12.3 Não liberar memória alocada dinamicamente quando ela não mais for necessária pode fazer com que o sistema esgote prematuramente sua memória. Algumas vezes isso é chamado um "vazamento de memória".

12.4 Liberar memória não alocada dinamicamente com **malloc.**

12.5 Fazer referência à memória que foi liberada.

12.6 Não definir como **NULL** o link na base de uma pilha.

12.7 Não definir como **NULL** o link no último nó dc uma fila.

12.8 Não definir como **NULL** os links nos nós folhas de uma árvore.

Práticas Recomendáveis de Programação

12.1 Use o operador **sizeof** para determinar o tamanho de uma estrutura.

12.2 Ao usar **malloc,** faça a verificação se o valor de retorno do ponteiro é **NULL.** Imprima uma mensagem de erro se a memória solicitada não foi alocada.

12.3 Quando a memória que foi alocada dinamicamente não for mais necessária, use **f ree** para devolver imediatamente a memória ao sistema.

12.4 Atribuir **NULL** ao membro de ligação (link) de um novo nó. Os ponteiros devem ser inicializados antes de serem usados.

Dicas de Performance

12.1 Um array pode ser declarado de forma a conter mais elementos do que o número esperado de itens de dados, mas isso pode desperdiçar memória. As listas encadeadas permitem utilização melhor da memória nessas situações.

12.2 A inserção e a eüminação (remoção) em um array ordenado podem ser demoradas — todos os elementos a partir do elemento inserido ou removido devem ser deslocados adequadamente.

12.3 Os elementos de um array podem ser armazenados contiguamente na memória. Isso permite acesso imediato a qualquer elemento do array porque o endereço de qualquer elemento pode ser calculado diretamente com base em sua posição relativa ao início do array. As listas encadeadas não permitem tal acesso imediato aos seus elementos.

12.4 Usar alocação dinâmica de memória (em vez de arrays) para estruturas de dados que aumentam e diminuem em tempo de execução pode economizar memória. Entretanto, tenha em mente que os ponteiros ocupam espaço e que a memória alocada dinamicamente oferece o risco de overhead de chamadas de funções.

Dica de Portabilidade

12.1 O tamanho de uma estrutura não é necessariamente a soma dos tamanhos de seus membros. Isso acontece devido às várias exigências de alinhamento de limites que diferem de um equipamento para outro (veja o Capítulo 10).

Exercícios de Revisão

12.1 Preencha as lacunas a seguir:

a) Uma estrutura auto-_é usada na criação de estruturas dinâmicas de dados.

b) A função_é usada para alocar memória dinamicamente.

c) Uma_é uma versão especializada de uma lista encadeada na qual os nós só podem ser inseridos e removidos no início da lista.

d) As funções que não modificam uma lista encadeada, mas simplesmente servem para sua verificação são conhecidas como_.

e) Uma fila é conhecida como uma estrutura de dados_porque os primeiros nós inserido os primeiros a serem removidos.

f) O ponteiro para o próximo nó em uma lista encadeada é conhecido como_.

g) A função_é usada para recuperar a memória alocada dinamicamente.

h) Uma_é uma versão especializada de uma lista encadeada na qual os nós só podem ser inseridos no início de uma lista e removidos do final da lista.

i) Uma_é uma estrutura de dados bidimensional e não-linear que contém nós com dois ou mais links.

j) Uma pilha é conhecida como uma estrutura de dados_porque o último nó inserido é o primeiro a ser removido.

k) Os nós de uma árvore_contêm dois membros de ligação.

l) O primeiro nó de uma árvore é o nó_.

m) Cada link de um nó da árvore aponta para um_ou uma_daquele nó.

n) Em uma árvore, um nó que não possua filhos é chamado nó_.

o) Os algoritmos de travessia (percurso) de uma árvore binaria são_,_e_

12.2 Quais as diferenças entre uma lista encadeada e uma pilha?

12.3 Quais as diferenças entre uma pilha e uma fila?

12.4 Escreva uma instrução ou um conjunto de instruções para realizar cada um dos pedidos a seguir. Admita que todas as manipulações acontecem em **main** (portanto, não há necessidade de endereços de variáveis ponteiros) e admita as seguintes definições:

struct noGrau {
char sobrenome[20]; float grau;
struct noGrau *proximoPtr;
};
typedef struct noGrau NOGRAU; typedef NOGRAU *NOGRAUPTR;

a) Crie um ponteiro para o início da lista chamado **inicioPtr.** A lista está vazia.

b) Crie um novo nó do tipo **NOGRAU** que esteja apontado pelo ponteiro **novoPtr** do tipo **NOGRAUPTR.** Atribua a string " **Jarbas**" ao membro **sobrenome** e o valor **91. 5** ao membro **grau** (use **strcpy).** Forneça quaisquer declarações e instruções necessárias.

c) Assuma que a lista para a qual **inicioPtr** aponta consiste atualmente em 2 nós — um contendo **" Jarbas**" e um contendo **"Si 1 va**". Os nós estão em ordem alfabética. Forneça as instruções necessárias para inserir nós em ordem alfabética que contenham os seguintes dados para **sobrenome** e **grau:**

"Alves" 85.0 "Torres" 73.5 "Pereira" 66.5

Use os ponteiros **anteriorPtr, atualPtr** e **novoPtr** para realizar as inserções. Declare para o que **anteriorPtr** e **atualPtr** apontam antes de cada inserção. Assuma que **novoPtr** aponta

sempre para um novo nó e que os dados já foram atribuídos ao novo nó.

d) Escreva um loop **while** que imprima os dados em cada nó da lista. Use o ponteiro **atualPtr** para se mover ao longo da lista.

e) Escreva um loop **whi le** que remova todos os nós da lista e libere a memória associada a cada nó. Use os ponteiros **atualPtr** e **tempPtr** para percorrer a lista e liberar memória, respectivamente.

12.5 Forneça manualmente os percursos inorder, preorder e postorder da árvore binaria da Fig. 12.22.

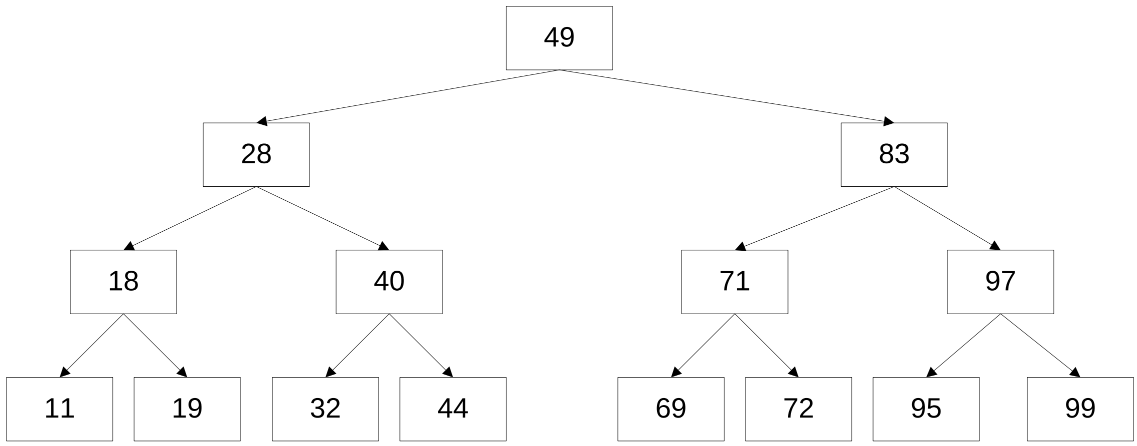


Fig. 12.22 Uma árvore binária de pesquisa com 15 nós.

Respostas dos Exercícios de Revisão

12.1 a) referenciada, b) **malloc.** c) pilha, d) predicadas, e) FIFO. f) link. g) **f ree.** h) fila. i) árvore, j) LIFO. k)binaria. **1)** raiz m) filho ou subárvore. n) folha, o) inorder, preorder, postorder.

12.2 E possível inserir um nó em qualquer lugar de uma lista encadeada, assim como remover um nó de qualquer lugar de uma lista encadeada. Entretanto, em uma pilha, só podem ser inseridos nós no topo da pilha e só podem ser removidos nós do topo da pilha.

12.3 Uma fila tem ponteiros tanto para seu início quanto para seu final, para que os nós possam ser inseridos no final e removidos do início. Uma pilha tem um único ponteiro para o topo, onde são realizadas a inserção e a remoção de nós.

12.4 a) **NOGRAUPTR inicioPtr = NULL;**

b) **NOGRAUPTR novoPtr;**

novoPtr = malloc(sizeof(NOGRAU)); strcpy(novoPtr->sobrenome, "Jarbas"); novoPtr->grau = 91.5; novoPtr->proximoPtr = NULL;

c) Para inserir **"Alves":**

anteriorPtr é **NULL, atualPtr** aponta para o primeiro elemento da lista. **novoPtr->proximoPtr = atualPtr; inicioPtr = novoPtr;**

Para inserir **"Torres":**

anteriorPtr aponta para o último elemento da lista (contendo **"Silva") atualPtr**

é **NULL.**

novoPtr->proximoPtr = atualPtr; anteriorPtr->proximoPtr = novoPtr;

Para inserir **"Pereira":**

anteriorPtr aponta para o nó contendo " **Jarbas" atualPtr** aponta para o nó contendo " **Silva". novoPtr->proximoPtr = atualPtr; anteriorPtr->proximoPtr = novoPtr;**

d) **atualPtr = inicioPtr; while (atualPtr != NULL) {**
printf("Sobrenome = %s\nGrau = %6.2f\n",
atualPtr->sobrenome, atualPtr->grau); atualPtr = atualPtr->proximoPtr;
}

e) **atualPtr = inicioPtr; while (atualPtr != NULL) {**
tempPtr = atualPtr;
atualPtr = atualPtr->proximoPtr;
free(tempPtr);
>
inicioPtr = NULL;

12.5 O percurso inorder é:
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99 O percurso preorder é:
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99 O percurso postorder é:
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Exercícios

12.6 Escreva um programa que concatene duas listas encadeadas de caracteres. O programa deve incluir a função **concatenar** que utilize ponteiros para ambas as listas como argumentos e concatene a segunda lista com a primeira.

12.7 Escreva um programa que una duas listas ordenadas de inteiros em uma única lista ordenada. A função **unir** deve receber ponteiros para o primeiro nó de cada uma das listas a serem unidas e deve retornar um ponteiro para o primeiro nó da lista resultante.

12.8 Escreva um programa que insira, em ordem, 25 números inteiros aleatórios de 0 a 100 em uma lista encadeada. O programa deve calcular a soma dos elementos como um número inteiro e a média dos elementos como um número de ponto flutuante.

12.9 Escreva um programa que crie uma lista encadeada de 10 caracteres e então crie uma cópia da lista na ordem inversa.

12.10 Escreva um programa que receba uma linha de texto e use uma pilha para imprimir a linha invertida.

12.11 Escreva um programa que use uma pilha para determinar se uma string é um palíndromo (i.e., a string é soletrada identicamente nos sentidos normal e inverso). O programa deve ignorar espaços e pontuação.

12.12 As pilhas são usadas por compiladores para ajudar no processo de calcular expressões e gerar código em linguagem de máquina. Neste e no próximo exercício, investigamos

como os compiladores calculam ex-pressões aritméticas que consistem apenas em constantes, operadores e parênteses.

Os seres humanos geralmente escrevem expressões como 3 + 4 e 7 / 9 nas quais o operador (+ ou / aqui) é escrito entre seus operandos — isso é chamado *notação infixada (infix).* Os computadores 1 "preferem" a *notação posfixada (postfix)* na qual o operador é escrito à direita de seus dois operandos. As expressões infixas anteriores apareceriam em notação posfixada como 3 4 + e7 9 /.respectivamente.

Para calcular uma expressão infixada complexa, um compilador deveria converter inicialmente a expressão para notação posfixada e depois calcular a versão posfixada da expressão. Cada um desses algoritmos exige apenas uma única passada da esquerda para a direita na expressão. Cada algoritmo usa uma pilha em suporte à sua operação e em cada um deles a pilha é usada com um objetivo diferente.

Neste exercício, você escreverá uma versão em C do algoritmo de conversão infixada-para-posfixada. No exercício seguinte, você escreverá uma versão em C do algoritmo de cálculo da expressão.

Escreva um programa que converta uma expressão aritmética comum infixada (assuma que foi fornecida uma expressão válida) com dígitos inteiros simples como

(6+2) *5-8/4

em uma expressão posfixada. A versão posfixada da expressão infixada anterior é 62 + 5*84/-

O programa deve ler a expressão no array de caracteres **infix** e usar versões modificadas das funções de pilhas implementadas neste capítulo para auxiliar a criação da expressão posfixada no array de caracteres postfix. O algoritmo para a criação da expressão posfixada é o seguinte:

1) Coloque um parêntese esquerdo ' (' na pilha.

2) Acrescente um parêntese direito ') ' no final de **infix.**

3) Enquanto a pilha não estiver vazia, leia **infix** da esquerda para a direita e faça o seguinte:

Se o caractere atual em infix for um dígito, copie-o para o próximo elemento de postfix.

Se o caractere atual em infix for um parêntese esquerdo, coloque-o na pilha.

Se o caractere atual em infix for um operador,

Remova os operadores (se houver algum) no topo da pilha enquanto eles tiverem precedência maior ou igual à do operador atual e insira os operadores removidos em postfix. Coloque na pilha o caractere atual em infix.

Se o caractere atual em infix for um parêntese direito

Remova os operadores do topo da pilha e insira-os em **postf ix** até que um parêntese esquerdo esteja no topo da pilha.

Remova (e elimine) da pilha o parêntese esquerdo.

Os seguintes operadores aritméticos são permitidos em uma expressão: + adição

- subtração

* multiplicação

/ divisão *

^A exponenciação *

% resto (modulus)

A pilha deve ser conservada por meio das seguintes declarações: **struct noPilha { char dados;**

struct noPilha *proximoPtr;

>;

typedef struct noPilha NOPILHA; typedef NOPILHA *NOPILHAPTR;

O programa deve consistir em **main** e oito outras funções com os seguintes cabeçalhos: **void converteParaPostfix(char infix[], char postfix[])**

Converte a expressão infixada para a notação posfixada. **int seOperador(char c)**

Determina se **c** é um operador, **int precedência(char operadorl, char operador2)**

Determina se a precedência de **operadorl** é menor, igual ou maior do que a de **operador2.**

A função retorna — 1, 0 e 1, respectivamente, **void push(NOPILHAPTR *topoPtr, char valor)**

Coloca um valor na pilha, **char pop(NOPILHAPTR *topoPtr)**

Remove um valor da pilha, **char topoPilha(NOPlLHAPTR topoPtr)**

Retorna o valor do topo da pilha sem retirá-lo. **int estaVazia(NOPILHAPTR topoPtr)**

Determina se a pilha está vazia, **void imprimePilha(NOPILHAPTR topoPtr)**

Imprime a pilha.

12.13 Escreva um programa que calcule uma expressão posfixada (admita que ela é válida) como **62 + 5*84/-**

O programa deve ler em um array de caracteres uma expressão posfixada consistindo em dígitos e operadores. Usando versões modificadas das funções de pilhas implementadas anteriormente neste capítulo, o programa deve examinar a expressão e calculá-la. O algoritmo é o seguinte:

1) Coloque o caractere **NULL (' \ 0** ') no final da expressão posfixada. Quando o caractere **NULL** for encontrado, não se faz necessário mais nenhum processamento.

2) Enquanto ' \ **0** ' não for encontrado, leia a expressão da esquerda para a direita.

Se o caractere atual for um dígito.

coloque seu valor inteiro na pilha (o valor inteiro de um caractere é seu valor no conjunto de caracteres menos o valor de ' **0** ' no conjunto de caracteres do computador). Caso contrário, se o caractere atual for um operador.

Retire os dois elementos do topo da pilha e coloque-os nas variáveis **x** e **y.**

Calcule **y operador x.**

Coloque o resultado do cálculo na pilha.

3) Quando o caractere **NULL** for encontrado na expressão, retire o valor do topo da pilha. Esse é o resultado da expressão posfixada.

Nota: Em 2), se o operador for ' / ', o topo da pilha for **2** e o próximo elemento na pilha for **8,** então coloque **2** em **x, 8** em **y,** calcule **8 / 2** e coloque o resultado, **4,** de volta na pilha. Essa nota também se aplica ao operador ' - '. São as seguintes as operações aritméticas permitidas em uma expressão:

+ adição

- subtração

* multiplicação / divisão ^ exponenciação % resto (modulus)

A manutenção da pilha deve ser realizada por meio das seguintes declarações:

struct noPilha { int dados;
struct noPilha *proximoPtr;
};
typedef struct noPilha NOPILHA; typedef NOPILHA *NOPILHAPTR;

O programa deve consistir em **main** e seis outras funções com os seguintes cabeçalhos:

int calculaExpressaoPosfixada(char *expr)
Calcula a expressão posfixada.
int calcula(int opl, int op2, char operador)
Calcula a expressão **opl operador op2.**
void push(NOPILHAPTR *topoPtr, int valor)
Coloca um valor na pilha,
int pop(NOPILHAPTR *topoPtr)
Retira um valor da pilha,
int estaVazia(NOPILHAPTR topoPtr)
Determina se a pilha está vazia,
void imprimePilha(NOPILHAPTR topoPtr)
Imprime a pilha.

12.14 Modifique o programa que calcula a expressão posfixada no Exercício 12.13 de modo que ele possa processar operandos inteiros maiores do que 9.

12.15 *(Simulação de Supermercado)* Escreva um programa que simule a fila de clientes no caixa de um super-mercado. Os clientes devem ser programados em uma fila e chegam em intervalos de 1 a 4 minutos. Além disso, cada cliente é atendido em intervalos inteiros aleatórios de 1 a 4 minutos. Obviamente, as taxas de chegada e de atendimento precisam ser equilibradas. Se a taxa média de chegada for maior do que a taxa média de atendimento, a fila crescerá indefinidamente. Mesmo com taxas equilibradas, o caráter aleatório da chegada e do atendimento ainda pode causar filas imensas. Execute o programa de simulação do super-mercado para um dia de 12 horas (720 minutos) usando o seguinte algoritmo:

1) Escolha um inteiro aleatório entre 1 e 4 para determinar o minuto no qual o primeiro cliente chega.

2) No momento da chegada do primeiro cliente:
Determine a hora de atendimento do cliente (inteiro aleatório entre 1 e 4); Comece a atender o cliente;
Marque a hora de chegada do próximo cliente (inteiro aleatório entre 1 e 4 adicionado à hora atual).

3) Para cada minuto do dia:
Se chegar o próximo cliente, Coloque-o na fila;
Marque o tempo dc chegada do próximo cliente: Se o atendimento do último cliente foi concluído;
Encerre seu atendimento
Tire da fila o próximo cliente a ser atendido
Determine o tempo de conclusão do atendimento^do cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).
Agora execute sua simulação para 720 minutos e responda a cada uma das perguntas a seguir:
a) Qual o número máximo de clientes na fila no período da simulação?
b) Qual a espera máxima experimentada por um dos clientes?
c) O que acontece se o intervalo de chegada for modificado de 1 a 4 minutos para 1 a 3 minutos?

12.16 Modifique o programa da Fig. 12.19 para permitir que a árvore binaria contenha valores duplicados.

12.17 Escreva um programa, baseado no programa da Fig. 12.19, que receba uma linha de texto, divida a sentença em palavras separadas, insira as palavras em uma árvore binaria e imprima os percursos inorder, preorder e postorder da árvore.

Sugestão: Leia a linha de texto em um array. Use **strtok** para dividir o texto em palavras. Quando uma divisão (palavra) for encontrada, crie um novo nó para a árvore, atribua o ponteiro retornado por **strtok** ao membro **string** do novo nó e insira o nó na árvore.

12.18 Neste capítulo, vimos que a eliminação de valores duplicados é simples durante a criação de uma árvore de pesquisa binaria. Descreva como você realizaria a eliminação de valores duplicados usando apenas um único array unidimensional. Compare o desempenho da eliminação baseada em array com a eliminação baseada em árvore de pesquisa binaria.

12.19 Escreva uma função **niveis** que receba uma árvore binaria e determine quantos níveis ela possui.

12.20 *(Imprime uma lista recursivamente no sentido inverso)* Escreva uma função **imprimeListaInversa** que imprima recursivamente os itens de uma lista na ordem inversa. Use sua função em um programa de teste que cria uma lista ordenada de inteiros e imprime a lista na ordem inversa.

12.21 *(Pesquisa recursivamente uma lista)* Escreva uma função **pesquisaLista** que pesquise recursivamente" uma lista encadeada, à procura de um valor específico. A função deve retornar um ponteiro para o valor se ele for encontrado; caso contrário, deve ser retornado **NULL.** Use sua função em um programa de teste que crie uma lista de inteiros. O programa deve pedir ao usuário um valor a ser localizado na lista.

12.22 *(Eliminação de árvore binaria)* Neste exercício, analisaremos a eliminação de itens de árvores de pesquisa binárias. O algoritmo de eliminação não é tão simples quanto o algoritmo de inserção. Há três casos que acontecem durante a eliminação de um item — o item está contido em um nó folha (i.e., não possui filhos), o item está contido em um nó que tem um filho ou o item está contido em um nó que tem dois filhos.

Se o item a ser removido estiver contido em um nó folha, o nó é eliminado e o ponteiro do nó pai é definido como **NULL.**

Se o item a ser removido estiver contido em um nó com um filho, o ponteiro no nó pai é definido para apontar para o nó filho, e o nó contendo o dado é eliminado. Isso faz com que o nó filho ocupe o lugar do nó eliminado na árvore.

O último caso é o mais difícil. Quando um nó com dois filhos é removido, outro nó da árvore deve tomar seu lugar. Entretanto, não se pode fazer simplesmente com que o ponteiro no nó pai aponte para um dos filhos do nó a ser eliminado. Na maioria dos casos, a árvore de pesquisa binaria resultante não respeitaria a seguinte característica das árvores de pesquisa binárias: *Os valores em qualquer subárvore esquerda são menores que o valor no nó pai, e os valores em qualquer subárvore direita são maiores que o*

valor no nó pai.

Que nó é usado como *nó de substituição* para conservar essa característica? Tanto o nó que contém o maior valor na árvore, menor do que o valor no nó que está sendo removido, como o nó que contém o menor valor na árvore, maior do que o valor no nó que está sendo removido. Vamos examinar o nó com o menor valor. Em uma árvore binaria, o maior valor, menor do que o valor de um nó pai, está localizado na subárvore esquerda do nó pai e certamente estará contido no nó situado na extremidade direita da subárvore. Esse nó é localizado percorrendo pela direita a subárvore da esquerda até que o ponteiro para o filho da direita do nó atual seja **NULL.** Agora estamos apontando para o nó de substituição que tanto é um nó folha como um nó com um filho à sua esquerda. Se o nó de substituição for um nó folha, os passos para realizar a eliminação são os seguintes:

1) Armazene o ponteiro para o nó a ser removido em uma variável ponteiro temporária (esse ponteiro é usado para liberar a memória alocada dinamicamente).

2) Defina o ponteiro no pai do nó a ser removido para apontar para o nó de substituição.

3) Defina o ponteiro no pai do nó de substituição como **NULL.**

4) Defina o ponteiro para a subárvore direita no nó de substituição para apontar para a subárvore direita do nó a ser removido.

5) Elimine o nó para o qual a variável ponteiro temporária aponta.

Os passos para eliminação de um nó de substituição com um filho à esquerda são similares aos de um nó de substituição sem filhos, mas o algoritmo também deve mover o filho para a posição do nó de substituição na árvore. Se o nó de substituição for um nó com um filho à esquerda, os passos para realizar a eliminação são os seguintes:

1) Armazene o ponteiro para o nó a ser removido em uma variável ponteiro temporária.

2) Defina o ponteiro no pai do nó a ser removido para apontar para o nó de substituição.

3) Defina o ponteiro no pai do nó de substituição para apontar para o filho à esquerda do nó de substituição.

4) Defina o ponteiro para a subárvore direita no nó de substituição para apontar para a subárvore direita do nó a ser removido.

5) Elimine o nó para o qual a variável ponteiro temporária aponta.

Escreva a função **eliminaNo** que utiliza como argumentos um ponteiro para o nó raiz da árvore e o valor a ser eliminado. A função deve localizar na árvore o nó que contém o valor a ser eliminado e usar os algoritmos analisados aqui para eliminar o nó. Se o valor não for encontrado na árvore, a função deve imprimir uma mensagem que indique se o valor foi eliminado ou não. Modifique o programa da Fig. 12.19 para usar essa função. Depois de eliminar um item, chame as funções de percurso **inOrder,** **preOrder** e **postorder** para confirmar que a operação de eliminação foi realizada corretamente.

12.23 *(Arvore de pesquisa binaria)* Escreva a função **arvorePesqBinaria** que tenta localizar um valor específico em uma árvore de pesquisa binaria. A função deve utilizar como argumento um ponteiro para o nó raiz da árvore binaria e um valor-chave de busca a ser localizado. Se o nó que contém o valor-chave de busca for encontrado, a função deve retornar um ponteiro para aquele nó; caso contrário, a função deve retornar um ponteiro **NULL.**

12.24 *(Percurso de árvore binaria por ordem de níveis)* O programa da Fig. 12.19 ilustrou três

métodos recursivos de percorrer uma árvore binaria — percursos inorder (in-ordem), preorder (pré-ordem) e postorder (pós-ordem). Este exercício apresenta o *percurso por ordem de níveis* de uma árvore binaria no qual os valores dos nós são impressos nível a nível, começando pelo nível do nó raiz. Os nós em cada nível são impressos da esquerda para a direita. O percurso por ordem de níveis não é um algoritmo recursivo. Ele usa a estrutura de dados em fila para controlar a saída dos nós. O algoritmo é o seguinte:

1) Insira o nó raiz na fila

2) Enquanto houver nós na fila,

Obtenha o próximo nó na fila Imprima o valor do nó

Se o ponteiro para o filho da esquerda não for **NULL**

Insira na fila o nó do filho da esquerda Se o ponteiro para o filho da direita não for **NULL**

Insira na fila o nó do filho direito.

Escreva a função **ordemNiveis** para realizar um percurso por ordem de níveis em uma árvore binaria. Modifique o programa da Fig. 12.19 para usar essa função. Compare a saída dessa função com as saídas dos outros algoritmos de percurso para ver se ela funcionou corretamente. (Nota: Você também precisará modificar e incorporar neste programa as funções de processamento de filas da Fig. 12.13.)

12.25 *(Imprimindo árvores)* Escreva uma função recursiva **saidaArvore** para exibir uma árvore binaria na tela. A função deve exibir a árvore, linha após linha, com o topo da árvore na esquerda da tela e o final da árvore no lado direito da tela. Cada linha é exibida verticalmente. Por exemplo, a árvore binaria ilustrada na Fig. 12.22 é exibida da seguinte maneira:

```
               99
          97
               92
     83
               72
          71
               69
49
               44
          40
               32
     28
               19
          18
               11
```

Observe que o nó da extremidade direita aparece no topo da saída, na coluna da extremidade direita, e que o nó raiz aparece no lado esquerdo da saída. Cada coluna da saída inicia cinco espaços à direita da coluna anterior. A função **saidaArvore** deve receber como argumentos um ponteiro para o nó raiz da árvore e um inteiro **totalEspacos** representando o número de espaços que antecedem o valor a ser exibido (essa variável deve iniciar com zero para que a exibição do nó raiz seja no lado esquerdo da tela). A função usa um percurso inorder modificado para exibir a árvore —

ele inicia com o nó da extremidade direita da tela e segue em direção à esquerda. O algoritmo é o seguinte:

Enquanto o ponteiro para o nó atual não for **NULL**

Chame recursivamente **saidaArvore** com a subárvore direita do nó atual e **totalEspacos** +5

Use uma estrutura **for** para contar de 1 a **totalEspacos** e exiba os espaços

Exiba o valor no nó atual

Defina o ponteiro para o nó atual para apontar para a subárvore esquerda do nó atual Incremente **totalEspacos** de 5.

Seção Especial: Construindo Seu Próprio Compilador

Nos Exercícios 7.18 e 7.19, apresentamos a Linguagem de Máquina Simpletron (LMS) e criamos o simulador de computador Simpletron para executar programas escritos em LMS. Nesta seção, construímos um compilador que converte os programas escritos em uma linguagem de programação de alto nível para LMS. Esta seção "une" todo o processo de programação. Escreveremos programas nessa linguagem de alto nível, compilaremos programas no compilador que construímos e executaremos programas no simulador construído no Exercício 7.19.

12.26 *(A Linguagem Simples)* Antes de começar a construir o compilador, analisaremos uma linguagem de alto nível simples, porém poderosa, similar às primeiras versões da popular linguagem BASIC. Chamamos a linguagem *Simples.* Toda *sentença* da linguagem Simples consiste em um *número de Unha* e uma *instrução.* Os números de linhas devem aparecer na ordem ascendente. Cada instrução começa com um dos seguintes *comandos:* **rem, input, let, print, goto, if/goto** ou **end** (veja a Fig. 12.23). Todos os comandos, exceto **end,** podem ser usados repetidamente. A linguagem Simples calcula apenas expressões inteiras usando os operadores +, -, * e /. Esses operadores possuem a mesma precedência que na linguagem C. Os parênteses podem ser usados para modificar a ordem de cálculo de uma expressão.

Nosso compilador Simples reconhece apenas letras minúsculas. Todos os caracteres em um arquivo Simples devem constar de letras minúsculas (letras maiúsculas resultam em um erro de sintaxe, a menos que apareçam em uma sentença iniciada com **rem,** caso em que são ignoradas). Um *nome de variável é* uma única letra. A linguagem Simples não permite nomes de variáveis descritivos, portanto, essas devem ser explicadas em comentários que indiquem seu uso no programa. A linguagem Simples usa apenas variáveis inteiras e não possui declaração de variáveis — a simples menção de um nome de variável em um programa faz com que ela seja declarada e inicializada automaticamente com o valor zero. A sintaxe da linguagem Simples não permite manipulação de strings (ler uma string, escrever uma string, comparar strings etc). Se for encontrada uma string em um programa Simples (depois de um comando diferente de **rem),** o compilador gera um erro de sintaxe. Nosso compilador assumirá que os programas em Simples foram digitados corretamente. O Exercício 12.29 pede ao aluno que modifique o compilador para realizar a verificação de erros de sintaxe.

Comando	Exemplo de sentença	Descrição
rem	**50 rem isso e um comentário**	Qualquer texto após o comando **rem** serve apenas para a documentação e é ignorado pelo compilador
input	**30 input x**	Exibe um ponto de interrogação para pedir ao usuário que digite um inteiro. Lê tal inteiro do teclado e armazena-o em x.
let	**80 let u = 4 * (j – 46)**	Atribui a **u** o valor de **4 * (j – 56).** Observe que a expressão arbitrária complexa pode aparecer à direira do sinal de igual.
print	**10 print w**	Exibe o valor de **w.**
goto	**70 goto 45**	Transfere o controle do programa para a linha **45.**
if / goto	**35 if i == z goto 80**	Verifica se **i** é igual a **z** e transfere o controle do programa para a linha **80** se a condição for verdadeira; caso contrário, continua a execução com a próxima sentença.
end	**99 end**	Encerra a execução do programa

Fig. 12.23 Comando da linguagem Simples.

```
10    rem determina e imprime a soma de dois inteiros
15    rem
20    rem recebe os dois inteiros
30    input a
40    input b
45    rem
```

```
50    rem soma os inteiros e armazena o resultado em c
60    let c = a + b
65    rem
70    rem imprime o resultado
80    print c
90    rem termina a execução do programa
99    end
```

Fig. 12.24 Determina a soma de dois inteiros.

A linguagem Simples usa a instrução condicional if/gotoea instrução incondicional goto para alterar o fluxo de controle durante a execução de um programa. Se a condição na instrução if/goto for verdeira, o controle é transferido para uma linha específica do programa. Os seguintes operadores relacionais e de igualdade são válidos em uma instrução if/goto: <, >, <=, >=, == ou ! =. A precedência desses operadores é a mesma do C.

Vamos examinar vários exemplos de programas na linguagem Simples que demonstram seus recursos. O primeiro programa (Fig. 12.24) lê dois inteiros do teclado, armazena os valores nas variáveis a e b e calcula e imprime sua soma (armazenada na variável c).

O programa da Fig. 12.25 determina e imprime o maior de dois inteiros. Os inteiros são fornecidos por meio do teclado e armazenados em **s** e t. A instrução if/goto verifica a condição **s** >= t. Se a condição for verdadeira, o controle é transferido para a linha **90** e **s** é impresso; caso contrário, t é impresso e o controle é transferido para a instrução end na linha **99,** onde o programa termina.

A linguagem Simples não fornece uma estrutura de repetição (como as estruturas for, whi le ou do/while). Entretanto, a linguagem Simples pode simular cada uma das estruturas de repetição do C usando as instruções if/goto e goto. A Fig. 12.26 usa um loop controlado por sentinela para calcular o quadrado de vários inteiros. Cada inteiro é fornecido por meio do teclado e armazenado na variável j. Se o valor fornecido for o sentinela **-9999**, o controle é transferido para a linha **99** quando o programa termina. Caso contrário, o quadrado de j é atribuído a k, k é impresso na tela e o controle é passado para a linha **2**0, onde o próximo inteiro é recebido.

Usando os programas das Figs. 12.24, 12.25 e 12.26 como guia, escreva um programa na linguagens Simples que realize cada uma das seguintes tarefas:

```
10    rem    determina o maior entre dois inteiros
20    input s
30    input t ^
32    rem
35    rem    verifica se s >= t
40    if s >= t goto 90
45    rem
50    rem    t e maior que s, portanto imprime t
60    print t
70    goto 99
75    em
80    rem    s e maior ou igual a t, portanto imprime s
90    print s
99    end
```

Fig. 12.25 Encontra o maior de dois inteiros,

```
10   rem    calcula o quadrado de vários inteiros
20   input j
23   rem
25   rem    verifica o valor sentinela
30   if j  ==  -9999 goto 99
33   rem
35   rem    calcula o quadrado de j e atribui o resultado a k
40   let k = j * j
50   print k ^
53   rem
55   rem   loop para obter o próximo j
60   goto 2 0
99   end
```

Fíg. 12.26 Calcula o quadrado de vários inteiros.

a) Receba três números inteiros, calcule sua média e imprima o resultado.

b) Use um loop controlado por sentinela para receber 10 inteiros e calcule e imprima sua soma.

c) Use um loop controlado por contador para receber 7 inteiros, alguns positivos e outros negativos, e calcule e imprima sua média.

d) Receba uma série de inteiros e determine e imprima o maior. O primeiro inteiro recebido indica quantos números devem ser processados.

e) Receba 10 inteiros e imprima o menor.

f) Calcule e imprima a soma dos inteiros pares de 2 a 30.

g) Calcule e imprima o produto dos inteiros ímpares de 1 a 9.

12.27 *(Construindo um Compilador; Pré-requisito: Exercícios Completos7.18, 7.19, 12.12,12.13 e 12.26)* Agora que a linguagem Simples foi apresentada (Exercício 12.26), analisaremos como construir nosso compilador Simples. Em primeiro lugar, examinamos o processo pelo qual um programa Simples é convertido para a LMS e executado pelo simulador Simpletron (veja a Fig. 12.27). Um arquivo contendo um programa Simples é lido pelo compilador e convertido para o código LMS. O código LMS é enviado para um arquivo em disco, no qual aparece uma instrução LMS por linha. O arquivo LMS é então carregado no simulador Simpletron e os resultados são enviados para um arquivo em disco e para a tela. Observe que o programa Simpletron desenvolvido no Exercício 7.19 recebia dados do teclado. Ele deve ser modificado para ler dados de um arquivo para que possa executar os programas produzidos por nosso compilador.

O compilador realiza duas *passadas* do programa Simples para convertê-lo a LMS. A primeira passada constrói uma *tabela de símbolos* na qual todos os *números de linhas*, *nomes de variáveis* e *constantes* do programa na linguagem Simples são armazenados com seu tipo e localização correspondente no código LMS final (a tabela de símbolos é analisada detalhadamente a seguir). A primeira passada também produz as instruções LMS correspondentes para cada instrução em Simples. Como veremos, se o programa em Simples possuir instruções que transferem o controle para uma linha posterior do programa, a primeira passada resulta em um programa LMS contendo algumas instruções incompletas. A segunda passada do compilador localiza e completa as instruções inacabadas e envia o programa LMS para um arquivo.

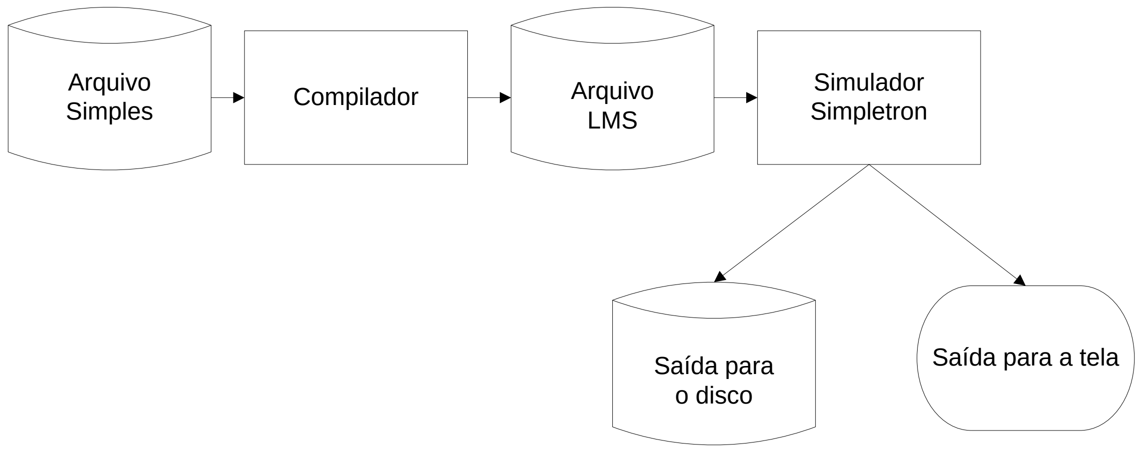


Fig. 12.27 Escrevendo, compilando e executando um programa Simples.

Primeira Passada

O compilador começa lendo uma sentença do programa na linguagem Simples para a memória. A linha deve ser separada em suas "partes" (ou "tokens", i.e., em "pedaços" de uma sentença) para processamento e compilação (a função **strtok** da biblioteca padrão pode ser usada para facilitar essa tarefa.) Lembre-se de que todas as instruções começam com um número de linha seguido de um comando. Quando o compilador divide uma sentença em partes, elas serão colocadas na tabela de símbolos se forem um número de linha, uma variável ou uma constante. Um número de linha só será colocado na tabela de símbolos se for a primeira parte de uma sentença. A **tabelaSimbolos** é um array de estruturas **entradaTabela** que representam cada símbolo do programa. Não há restrições quanto ao número de símbolos que podem aparecer em um programa. Portanto, **tabelaSimbolos** de um determinado programa pode ser grande. Por ora, faça com que **tabelaSimbolos** seja um array de 100 elementos. Você pode aumentar ou diminuir seu tamanho depois que o programa estiver funcionando.

A definição da estrutura **entradaTabela** é a seguinte:

```
struct entradaTabela { int simbolo;
char tipo;  /*   ' C,   ' L'  ou  'V */ int local;  /* 00 a 99 */
}
```

Cada estrutura **entradaTabela** contém três membros. O membro **simbolo** é um inteiro que contém a representação ASCII de uma variável (lembre-se de que os nomes de variáveis são caracteres isolados), um número de linha ou uma constante. O membro **tipo** é um dos seguintes caracteres que indica o tipo do símbolo: ' **C** ' para uma constante, ' **L** ' para um número de linha ou ' **V**' para uma variável. O membro **local** contém o local da memória do Simpletron **(00** a **99)** à qual o símbolo se refere. A memória do Simpletron é um array de 100 inteiros no qual as instruções LMS e os dados são armazenados. Para um número de linha, o local é o elemento no array da memória do Simpletron na qual iniciam as instruções LMS para a sentença em linguagem Simples. Para uma variável ou uma constante, o local é o elemento no array da memória do Simpletron no qual a variável ou constante está armazenada. As variáveis e constantes são alocadas do final da memória do Simpletron para a frente. A primeira variável ou constante é armazenada no local **99,** a próxima, no local **98** etc.

A tabela de símbolos desempenha um papel importante na conversão de programas na linguagem Simples para LMS. Aprendemos no Capítulo 7 que uma instrução em LMS é um inteiro de quatro dígitos composto de duas partes — o *código de operação* e o *operando.* O código de operação é determinado pelos comandos em Simples. Por exemplo, o comando **input** da linguagem Simples corresponde ao

código de operação **10** (read, ou ler) e o comando **print** da linguagem Simples corresponde ao código de operação **11** (write, ou escrever). O operando é um local da memória que contém os dados nos quais o código da operação realiza sua tarefa (e.g., o código de operação **10** lê um valor do teclado e armazena-o no loca da memória especificado pelo operando). O compilador consulta **tabelaSimbolos** para determinar o local da memória de Simpletron de cada símbolo para que o local correspondente possa ser usado para comple- J tar as instruções LMS.

A compilação de cada instrução da linguagem Simples se baseia em seu comando. Por exemplo, depois de o número de linha em uma instrução **rem** ser inserido na tabela de símbolos, o restante da instrução é ignorado pelo compilador porque um comentário só tem a finalidade de documentar o programa. As instruções **input, print, goto** e **end** correspondem às instruções *read, write, branch* (para um local específico) e *halt.* As instruções que possuírem esses comandos da linguagem Simples são convertidas diretamente em LMS (observe que a instrução **goto** pode conter uma referência indeterminada se o número de linha especificado se referir a uma instrução mais adiante no arquivo de programa Simples; algumas vezes isso é chamado referência antecipada).

Quando uma instrução **goto** é compilada com uma referência indeterminada, a instrução LMS deve ser *marcada (sinalizada,* ou *flagged)* para indicar que a segunda passada do compilador deve completar a instrução. Os sinalizadores são armazenados no array **f lags** de 100 elementos do tipo **int,** no qual cada elemento é inicializado com **-1.** Se o local da memória, ao qual o número da linha em um programa Simples se refere, ainda não for conhecido (i.e., não estiver na tabela de símbolos), o número da linha é armazenado no array **f lags** no elemento com o mesmo subscrito que a instrução incompleta. O operando da instrução incompleta é definido temporariamente como **00.** Por exemplo, uma instrução de desvio incondicional (fazendo uma referência antecipada) é deixada como **+4 000** até a segunda passada do compílador. Em breve será descrita a segunda passada do compilador.

A compilação das instruções **if /goto** e **let** é mais complicada que outras instruções — elas são as únicas instruções que produzem mais de uma instrução LMS. Para uma instrução **if/goto,** o compilador produz código para examinar a condição e para desviar para outra linha, se necessário. O resultado do desvio pode ser uma referência indeterminada. Cada um dos operadores relacionais e de igualdade pode ser simulado usando as instruções de *desvio zero* e *desvio negativo* da LMS (ou possivelmente uma combinação de ambas).

Para uma instrução **let,** o compilador produz código para calcular uma expressão aritmética complexa consistindo em variáveis inteiras e/ou constantes. As expressões devem separar cada operando e operador por meio de espaços. Os Exercícios 12.12 e 12.13 apresentaram o algoritmo de conversão infixada-para-posfixada e o algoritmo de cálculo posfixado usado por compiladores na avaliação de expressões. Antes de prosseguir com nosso compilador, você deve completar cada um daqueles exercícios. Quando um compilador encontra uma expressão, ele,a converte da notação infixada para a notação posfixada e então calcula a expressão.

Como o compilador produz linguagem de máquina para calcular uma expressão que contém variáveis?

O algoritmo de cálculo posfixado contém uma "conexão" que permite ao nosso compilador gerar instruções LMS em vez de realmente calcular a expressão. Para possibilitar a existência dessa "conexão" no compilador, o algoritmo de cálculo posfixado deve ser modificado para pesquisar na tabela de símbolos cada símbolo que encontrar (e possivelmente inseri-lo), determinar o local da memória correspondente àquele símbolo e *colocar na pilha o local da memória em vez do símbolo.* Quando um operador é encontrado em uma expressão posfixada, os dois locais da memória no topo da pilha são removidos e é produzida linguagem de máquina para realizar a operação, usando os locais da memória como operandos. O resultado de cada subexpressão é armazenado em um local temporário da memória e colocado novamente na pilha para que o cálculo da expressão posfixada possa continuar. Quando o

cálculo posfixado for concluído, o local da memória que contém o resultado é o único local que resta na pilha. Ele é removido e são geradas as instruções LMS para atribuir o resultado à variável à esquerda da instrução **let.**

Segunda Passada

A segunda passada do compilador realiza duas tarefas: determinar todas as referências indeterminadas e enviar o código LMS para um arquivo. A determinação das referências ocorre da seguinte maneira:

1) Procure no array **flags** uma referência indeterminada (i.e., um elemento com valor diferente de **-1).**

2) Localize no array **tabelaSimbolos** a estrutura que contém o símbolo armazenado no array **flags** (certifique-se de que o tipo do símbolo é ' **L**' para um número de linha).

3) Insira o local da memória para o membro da estrutura **local** na instrução com a referência indeterminada (lembre-se de que uma instrução que contém uma referência indeterminada tem operando **00).**

4) Repita os passos 1, 2 e 3 até chegar ao fim do array **flags.**

Depois de o processo de resolução ser concluído, todo o array que contém o código LMS é enviado para um arquivo em disco com uma instrução LMS por linha. Esse arquivo pode ser lido pelo Simpletron para execução (depois de o simulador ser modificado para ler os dados de entrada a partir de um arquivo).

Um Exemplo Completo

O exemplo a seguir ilustra uma conversão completa de um programa Simples em LMS da forma como será realizado pelo compilador Simples. Considere um programa Simples que receba um inteiro e some os valores de 1 até aquele inteiro. O programa e as instruções LMS produzidas pela primeira passada são ilustrados na Fig. 12.28. A tabela de símbolos construída pela primeira passada é mostrada na Fig. 12.29.

A maioria das instruções Simples é convertida diretamente em instruções LMS. As exceções nesse programa são os comentários, a instrução **if /goto** na linha **20** e as instruções **let.** Os comentários não são traduzidos em linguagem de máquina. Entretanto, o número da linha de um comentário é colocado na tabela de símbolos, no caso de o número da linha ser referenciado por uma instrução **goto** ou **if/goto.** A linha **20** do programa especifica que, se a condição **y = = x** for verdadeira, o controle do programa é transferido para a linha **60.** Como a linha **60** aparece mais tarde no programa, a primeira passada do compilador ainda não colocou **60** na tabela de símbolos (os números de linhas são colocados na tabela de símbolos apenas quando aparecem como primeira parte ("token") de uma instrução). Portanto, não é possível, nesse momento, determinar o operando da instrução LMS de *desvio zero* no local **03** do array de instruções LMS. O compilador coloca **60** no local **03** do array **flags** para indicar que a segunda passada completa essa instrução.

Devemos controlar o local da próxima instrução no array LMS porque não há uma correspondência biunívoca entre as instruções Simples e as instruções LMS. Por exemplo, a instrução **if/goto** da linha **20** é compilada em três instruções LMS. Cada vez que uma instrução é produzida, devemos incrementar o *contador de instruções* para o próximo local do array LMS. Observe que o tamanho da memória Simpletron pode causar um problema para os programas Simples com muitas instruções, variáveis e constantes. É provável que o compilador fique sem memória. Para verificar esse caso, seu programa deve conter um *contador de dados* para controlar o local no qual a próxima variável ou constante será armazenada no array LMS.

Programa em Simples	**Instrução e Local**	**Descrição**

	LMS	
5 rem soma 1 a x	***nenhuma***	**00**
10 input x	**00 +1099**	**00**
15 rem verifica y == x	***nenhuma***	**99**
20 if y == x goto 60	**01 +2098**	**01**
	02 +3199	**01**
	03 +4200	**98**
25 rem incrementa y	***nenhuma***	**04**
30 let y = y + 1	**04 +2098**	**04**
	05 +3097	**97**
	06 +2196	**09**
	07 +2096	**09**
	08 +2198	**95**
35 rem soma y ao total	***nenhuma***	**14**
40 let t = t + y	**09 +2095**	**14**
	10 +3098	**15**
	11 +2194	**15**
	12 +2094	**16**
	13 +2195	
45 rem loop y	***nenhuma***	
50 goto 20	**14 +4001**	
55 rem imprime resultado	***nenhuma***	
60 print t	**15 +1195**	
99 end	**16 +4300**	

Fig. 12.28 Instrução LMS produzidas depois da primeira passada do compilador.

Se o valor do contador de instruções for maior que o valor do contador de dados, o array LMS está cheio.

Nesse caso, o processo de compilação deve terminar e o compilador deve imprimir uma mensagem de erro indicando que ficou sem memória durante a compilação.

Uma Apresentação Passo a Passo do Processo de Compilação

Vamos agora percorrer o processo de compilação do programa Simples da Fig. 12.28. O compilador lê a primeira linha do programa

5 rem soma 1 a x

na memória. A primeira parte da instrução (o número da linha) é determinada usando **strtok** (veja o Capítulo 8 para obter uma explicação sobre as funções de manipulação de strings). A parte ("token") retomada por **strtok** é convertida em um inteiro usando **atoi,** portanto o símbolo **5** pode ser colocado na tabela de símbolos. Se o símbolo não for encontrado, ele é inserido na tabela de símbolos. Como estamos no início do programa e essa é a primeira linha, ainda não há símbolos na tabela. Portanto, **5** é inserido na tabela de símbolos com o tipo **L** (número de linha) e é atribuído ao primeiro local do array LMS **(00).** Embora essa linha seja um comentário, um espaço na tabela de símbolos ainda é alocado para o número de linha (no caso de ela ser referenciada por um **goto** ou **if/goto).** Nenhuma instrução LMS é gerada por uma instrução **rem,** portanto o contador de instruções não é incrementado.

A seguir, a instrução

10 input x

Símbolo	**Tipo**	**Local**
5	**L**	**00**
10	**L**	**00**
‘x’	**V**	**99**
15	**L**	**01**
20	**L**	**01**
‘y’	**V**	**98**
25	**L**	**04**
30	**L**	**04**
1	**C**	**97**
35	**L**	**09**
40	**L**	**09**
‘t’	**V**	**95**
45	**L**	**14**
50	**L**	**14**
55	**L**	**15**
60	**L**	**15**
99	**L**	**16**

Fig. 12.29 Tabela de símbolos para o programa da Fig. 12.28.

é dividida em partes. O número de linha **10** é colocado na tabela de símbolos com o tipo **L** e é atribuído ao primeiro local do array LMS **(00** porque um comentário iniciou o programa e por isso o contador de instruções é atualmente **00).** O comando **input** indica que a próxima parte é uma variável (apenas uma variável pode aparecer em uma instrução **Input).** Como **input** corresponde diretamente a um código de operação LMS, o compilador simplesmente precisa determinar o local de **x** no array LMS. O símbolo **x** não é encontrado na tabela de símbolos. Dessa forma, ele é inserido na tabela de símbolos como a representação ASCII de **x,** recebe o tipo **V,** recebe o local **99** do array LMS (o armazenamento de dados começa em **99** e é alocado em sentido inverso). Agora pode ser gerado o código LMS para essa instrução. O código de operação **10** (o código de operação LMS para leitura) é multiplicado por 100 e o local de **x** (conforme a determinação da tabela de símbolos) é adicionado para completar a instrução. A instrução é então armazenada no array LMS no local **00.** O contador de instruções é incrementado de 1 porque uma única instrução LMS foi produzida.

A instrução

15 rem verifica y == x

é dividida a seguir. O número **15** é procurado na tabela de símbolos (e não é encontrado). O número da linha é inserido com o tipo ' **L'** e é atribuído à próxima posição no array, **01** (lembre-se de que as instruções **rem** não produzem código, portanto o contador de instruções não é incrementado).

A instrução

20 if y == x goto 60

é dividida a seguir. O número de linha **2 0** é inserido na tabela de símbolos e recebe o tipo **L** com a próxima posição no array LMS, **01.** O comando **if** indica que uma condição precisa ser verificada. A variável **y** não é encontrada na tabela de símbolos, portanto ela é inserida e recebe o tipo **V** e a posição **9 8** em LMS. A seguir, são geradas as instruções LMS para avaliar a condição. Como não há equivalência direta do **if/goto** com a LMS, aquele deve ser simulado realizando um cálculo que utiliza **x** e **y** e faz um desvio com base no resultado. Se **y** for igual a **x,** o resultado de subtrair **x** de **y** é zero, portanto pode ser usada a instrução de *desvio zero* com o resultado do cálculo para simular a instrução **if/goto.** O primeiro passo exige que **y** seja carregado (da posição **98** de LMS) no acumulador. Isto produzirá a instrução **01 +2098.** Em seguida, **x** *é* subtraído do acumulador. Isto produzirá a instrução **02 +3199.** O valor no

acumulador pode ser zero, positivo ou negativo. Como o operador é ==, queremos *desvio zero.* Em primeiro lugar, é procurado o local do desvio **(60,** nesse caso) na tabela de símbolos, que não é encontrado. Portanto, **60** colocado no array **flags,** no local **03,** e é gerada a instrução **03 +42 00** (não podemos adicionar o local do desvio porque ainda não atribuímos um local à linha **60** no array LMS). O contador de instruções é incrementado para **04.**

O compilador prossegue para a instrução

25 rem incrementa y

O número de linha **25** é inserido na tabela de símbolos com o tipo Leé atribuído ao local **04** de LMS. O contador de instruções não é incrementado. Quando a instrução

30 let y = y + 1

é dividida, o número de linha **3 0** é inserido na tabela de símbolos no local **04.** O comando **let** indica que a linha é uma instrução de atribuição. Primeiramente, todos os símbolos da linha são inseridos na tabela de símbolos (se já não existirem ali). O inteiro **1** é adicionado à tabela de símbolos com o tipo C e é atribuído à posição **97** de LMS. Em seguida, o lado direito da atribuição é convertido da notação infixada para posfixada. Depois disso, a expressão posfixada **(y 1 +)** é calculada. O símbolo **y** está presente na tabela de símbolos e sua posição correspondente na memória é colocada na pilha. O símbolo **1** também está presente na tabela de símbolos e sua posição de memória correspondente é colocada na pilha. Quando o operador + é encontrado, o calculador posfixado remove o valor do topo da pilha para o operando direito e remove outro valor do topo da pilha para o operando esquerdo, produzindo então as instruções LMS

04 +2098 ***(carrega y)***

05 +3097 *(soma 1)*

O resultado da expressão é armazenado em um local temporário da memória **(96)** com a instrução

06 +2196 *(armazena no local temporário)*

e o local temporário da memória é colocado na pilha. Agora que a expressão foi calculada, o resultado de ser armazenado em **y** (i.e., a variável no lado esquerdo do =). Assim sendo, o local temporário é carregado no acumulador e o acumulador é armazenado em **y** com as instruções

07 +2096 *(carrega do local temporário)*

08 +2198 *(armazena em y)*

O leitor observará imediatamente que as instruções LMS parecem ser redundantes. Analisaremos essa questão em breve.

Quando a instrução

35 rem soma y ao total

é dividida em partes, o número de linha **3 5** é inserido na tabela de símbolos com o tipo Leé atribuído ao local **09.**

A instrução

40 let t = t + y

é similar à linha **3 0**. A variável **t** é inserida na tabela de símbolos com o tipo **V** e é atribuída ao local **9 5 de** LMS. As instruções seguem a mesma lógica e formato da linha **3 0** e são geradas as instruções **09 +2095, 10 +3098,11 +2194,12 +2094 e l3 +2195.** Observe que o resultado de **t + v** é atribuído ao local temporário **94** antes de ser atribuído a **t(95).** Mais uma vez, o leitor observará que as instruções nos locais de memória **11** e **12** parecem ser redundantes. Analisaremos essa questão em breve.

A instrução

45 rem loop y

é um comentário, portanto a linha **4 5** é adicionada à tabela de símbolos com o tipo Leé atribuída ao local LMS 14

À instrução

50 goto 20

transfere o controle para a linha **2 0**. O número de linha **5 0** é inserido na tabela de símbolos com o tipo L e é atribuído ao local LMS **14.** O equivalente ao **goto** em LMS é a instrução de *desvio incondicional (40)* que transfere o controle para um local LMS específico. O compilador procura a linha **2 0** na tabela de símbolos e encontra que ele corresponde ao local LMS **01.** O código de operação **(4 0)** é multiplicado por 100 e o local **01** é adicionado a ele para produzir a instrução **14 +4001.**

A instrução

55 rem imprime resultado

é um comentário, portanto a linha 55 é inserida na tabela de símbolos com o tipo **L** e é atribuída ao local LMS 15.

A instrução

60 **print t**

é uma instrução de saída. O número da linha 6 0 é inserido na tabela de símbolos com o tipo **L** e é atribuído ao local LMS 15. O equivalente a **print** em LMS é o código de operação 11 *(yvrite).* O local de **t** é determinado a partir da tabela de símbolos e adicionado ao resultado do código da operação multiplicado por 100.

A instrução

99 **end**

é a linha final do programa. O número de linha 99 é armazenado na tabela de símbolos com o tipo **L** e é atribuído ao local LMS 16. O comando **end** produz a instrução LMS +4300 (43 é *halt* em LMS) que é escrita como a instrução final no array de memória LMS.

Isso completa a primeira passada do compilador. Agora vamos examinar a segunda passada. São procurados valores diferentes de -1 no array **f lags.** O local 03 contém 60, portanto o compilador sabe que a instrução 0 3 está incompleta. O compilador completa a instrução procurando por 6 0 na tabela de símbolos, determinando sua posição e adicionando o local à instrução incompleta. Nesse caso, a pesquisa determina que a linha 60 corresponde ao local LMS 15, assim a instrução completa 03 +4215é produzida, substituindo 03 +42 00. Agora o programa Simples foi compilado satisfatoriamente.

Para construir o compilador, você precisará realizar cada uma das seguintes tarefas:

a) Modifique o programa do simulador Simpletron escrito no Exercício 7.19 para receber dados de um arquivo especificado pelo usuário (veja o Capítulo 11). Além disso, o simulador deve enviar os resultados para um arquivo em disco no mesmo formato que a saída de tela.

b) Modifique o algoritmo de cálculo da notação infixada-para-posfixada do Exercício 12.12 para processar operandos inteiros com vários dígitos e operandos de nomes de variáveis com uma única letra. Sugestão: A função **strtok** da biblioteca padrão pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros usando a função **atoi** da biblioteca padrão. (Nota: A representação de dados da expressão posfixada deve ser alterada para suportar nomes de variáveis e constantes inteiras.)

c) Modifique o algoritmo de cálculo posfixado para processar operandos inteiros com vários dígitos e operandos de nomes de variáveis. Além disso, agora o algoritmo deve implementar a "conexão" mencionada anteriormente para que as instruções LMS sejam produzidas em vez de a expressão ser avaliada diretamente. Sugestão: A função **strtok** da biblioteca padrão pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros usando a função **atoi** da biblioteca padrão. (Nota: A representação de dados da expressão posfixada deve ser alterada para suportar nomes de variáveis e constantes inteiras).

d) Construa o compilador. Incorpore as partes (b) e (c) para calcular expressões em instruções **let.** Seu programa deve conter uma função que realize a primeira passada do compilador e outra função que realize a segunda passada do compilador. Ambas as funções podem chamar outras funções para realizar

suas tarefas.

12.28 *(Otimizando o Compilador Simples)* Quando um programa é compilado e convertido a LMS, é gerado um conjunto de instruções. Certas combinações de instruções repetem-se com freqüência, normalmente em grupos de três instruções chamados *produções.* Normalmente uma produção consiste em três instruções como *load (carregar), add (somar)* e *store (armazenar).* Por exemplo, a Fig. 12.30 ilustra cinco das instruções LMS produzidas na compilação do programa da Fig. 12.28. As três primeiras instruções são a produção que soma 1 a **y.** Observe que as instruções 06 e 07 armazenam o valor do acumulador no local temporário 96 e depois carregam novamente o valor no acumulador para que a instrução 0 8 possa armazenar o valor no local 9 8. Freqüentemente uma produção é seguida de uma instrução de carregamento para o mesmo local que acabou de ser armazenado. Esse código pode ser *otimizado* eliminando a instrução de armazenamento e a instrução de carregamento subseqüente que agem no mesmo local da memória. Essa otimização permitiria ao Simpletron executar o programa mais rapidamente porque há menos instruções nessa versão. A Fig. 12.31 ilustra a LMS otimizada para o programa da Fig. 12.28. Observe que há quatro instruções a menos no código otimizado — uma economia de 25% na memória.

04	**+2098**	**(carrega)**
05	**+3097**	**(soma)**
06	**+2196**	**(armazena)**
07	**+2096**	**(carrega)**
08	**+2198**	**(armazena)**

Fig. 12.30 Código não-otimizado do programa da Fig. 12,28.

Modifique o compilador para fornecer uma opção de otimizar o código da Linguagem de Máquina Simpletron que ele produz. Compare manualmente o código não-otimizado com o otimizado e calcule a porcentagem de redução.

12.29 *(Modificações no Compilador da Linguagem Simples)* Realize as seguintes modificações no compilador da linguagem Simples. Algumas dessas modificações também podem exigir modificações no programa do Simulador Simpletron escrito no Exercício 7.19.

a) Permita que o operador resto (ou modulus, %) seja usado em instruções **let.** A Linguagem de Máquina Simpletron deve ser modificada para incluir uma instrução com o operador resto.

b) Permita a exponenciação em uma instrução **let** usando ^A^ como operador de exponenciação. A Linguagem de Máquina Simpletron deve ser modificada para incluir uma instrução com o operador exponenciação.

c) Permita que o compilador reconheça letras maiúsculas e minúsculas em instruções da linguagem Simples (e.g., ' **A**' é equivalente a ' **a** ^1^**).** Não são exigidas modificações no Simulador Simpletron.

d) Permita que as instruções **input** leiam valores de muitas variáveis como **input x, y.** Não são exigidas modificações no Simulador Simpletron.

Programa em Simples			**Descrição**
5 rem soma 1 a x	**nenhuma**		**REM ignorado**
10 input x	**00**	**+1099**	**Le x no local 99**

15 rem verifica y == x	**Nenhuma**		**REM ignorado**
20 if y = x goto 60	**01**	**+2098**	**Carrega y(98) noacumulador**
	02	**+3199**	**Subtrai x(99) do acumulador**
	03	**+4211**	**Desvio para o local 11 se igual a zero**
25 rem incrementa y	**Nenhuma**		**REM ignorado**
30 let y=y +1	**04**	**+2098**	**Carrega y no acumulador**
	05	**+3097**	**Soma 1(97) ao acumulador**
	06	**+2198**	**Armazena o acumulador em y(98)**
35 rem soma y ao total	**Nenhuma**		**REM ignorado**
40 let t =t + y	**07**	**+2096**	**Carrega t do local(96)**
	08	**+3098**	**Soma y(98) ao acumulador**
	09	**+2196**	**Armazena o acumulador em t(96)**
45 rem loop y	**Nenhuma**		**REM ignorado**
50 goto 20	**10**	**+4001**	**Desvia para o local 01**
55 rem imprime resultado	**Nenhuma**		**REM ignorado**
60 print t	**11**	**+1196**	**Remete t(96) para a tela**
99 end	**12**	**+4300**	**Termina a execução**

Fig. 12.31 Código otimizado para o programa da Fig, 12.28,

e) Permita que o compilador imprima vários valores em uma única instrução **print** como **print a, b, c.** Não são exigidas modificações no Simulador Simpletron.

f) Adicione recursos de verificação de sintaxe ao compilador para que sejam exibidas mensagens de erro quando erros de sintaxe forem encontrados em um programa Simples. Não são exigidas modificações no Simulador Simpletron.

g) Permita arrays de inteiros. Não são exigidas modificações no Simulador Simpletron.

h) Permita sub-rotinas especificadas pelos comandos **gosub** e **return** na linguagem Simples. O comando **gosub** passa o controle do programa para uma sub-rotina e o comando **return** passa o controle de volta para a instrução, após o **gosub.** Isso é similar à função **call** em **C.** A mesma sub-rotina pode ser chamada de muitos **gosubs** distribuídos ao longo de um programa. Não são exigidas modificações no Simulador Simpletron.

i) Permita estruturas de repetição da forma

for x = 2 to 10 step 2
    Instruções em Simples
next

Essa instrução **for** faz um loop de **2** a **10** com incrementos **2.** A linha **next** marca o final do corpo da estrutura **for.** Não são exigidas modificações no Simulador Simpletron.

j) Permita estruturas de repetição da forma

for x = 2 to 10
    Instruções em Simples
next

Essa instrução **for** faz um loop de **2** a **10** com incremento default **1.** Não são exigidas modificações no Simulador Simpletron.

k) Permita que o compilador processe entrada e saída de strings. Isso exige que o Simulador Simpletron seja modificado para processar e armazenar valores de strings. Sugestão: Cada palavra do Simpletron pode ser dividida em dois grupos, cada um deles contendo um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente decimal ASCII de um caractere. Adicione uma instrução em linguagem de máquina que imprimirá uma string começando em um determinado local da memória do

Simpletron. A primeira metade da palavra naquele local é o total de caracteres na string (i.e., o comprimento da string). Cada meia palavra seguinte contém um caractere ASCII expresso em dois dígitos decimais. A instrução em linguagem de máquina verifica o comprimento e imprime a string traduzindo cada número de dois dígitos em seu caractere equivalente.

1) Permita que o compilador processe a adição de números de ponto flutuante a inteiros. O Simulador Simpletron também deve ser modificado para processar valores de ponto flutuante.

12.30 *(Um Interpretador da Linguagem Simples)* Um interpretador é um programa que lê uma instrução de outro programa em linguagem de alto nível, determina a operação a ser realizada pela instrução e executa a operação imediatamente. O programa não é convertido primeiro em linguagem de máquina. Os interpretadores são executados mais lentamente porque cada instrução encontrada no programa deve ser primeiramente decifrada. Se as instruções estiverem em um loop, elas são decifradas cada vez que forem encontradas no loop. As primeiras versões da linguagem de programação BASIC foram implementadas como interpretadores.

Escreva um interpretador para a linguagem Simples analisada no Exercício 12.26. O programa deve usar o conversor de notação infixada-para-posfixada desenvolvido no Exercício 12.12 e o calculador posfixado desenvolvido no Exercício 12.13 para calcular expressões em uma instrução **let.** As mesmas restrições impostas para a linguagem Simples no Exercício 12.26 devem ser respeitadas nesse programa. Teste o interpretador com os programas em Simples escritos no Exercício 12.26. Compare os resultados de executar esses programas no interpretador com os resultados de compilar programas em Simples e executá-los no simulador Simpletron construído no Exercício 7.19.

13

O Pré-processador

Objetivos

- Ser capaz de usar **#include** para desenvolver programas grandes.
- Ser capaz de usar **#def ine** para criar macros e macros com argumentos.
- Entender compilação condicional.
- Ser capaz de exibir mensagens de erro durante a compilação condicional.
- Ser capaz de usar assertivas para testar se os valores das expressões estão corretos.

Conserve a bondade; defina-a bem.
Alfred, Lord Tennyson

Encontrei-lhe um argumento; mas não sou obrigado a conseguir-lhe um entendimento.
Samuel Johnson

Um bom símbolo é o melhor argumento, e é um missionário para persuadir milhares de pessoas.
Ralph Waldo Emerson

As condições são fundamentalmente razoáveis.
Herbert Hoover (dezembro de 1929)

Um membro de partido, quando engajado em uma disputa, não se preocupa nada com os direitos de uma questão, apenas fica ansioso em convencer seus ouvintes de suas próprias assertivas.
Platão

Sumário

13.1 Introdução

Este capítulo apresenta o *pré-processador C.* O pré-processamento ocorre antes de um programa ser compilado. Algumas ações possíveis são: inclusão de outros arquivos no arquivo que está sendo compilado, definição de *constantes simbólicas* e *macros, compilação condicional* do código do programa e *execução condicional das diretivas do pré-processador.* Todas as diretivas do pré-processador começam com #, e somente os caracteres de espaço em branco podem aparecer antes de uma diretiva de pré-processador em uma linha.

13.2 A Diretiva #include do Pré-processador

A *diretiva **êinclude** do pré-processador* foi usada ao longo deste texto. A diretiva **#include** faz com que seja incluída uma cópia de um arquivo especificado no lugar da diretiva. As duas formas da diretiva **#include** são:

#include <nome-do-arquivo>
#include "nome-do-arquivo"

A diferença entre essas formas é o local no qual o pré-processador procura pelo arquivo a ser incluído.

Se o nome do arquivo estiver entre aspas, o pré-processador procura pelo arquivo a ser incluído no mesmo diretório do arquivo que está sendo compilado. Esse método é usado normalmente para incluir arquivos de cabeçalho definidos pelo usuário. Se o nome do arquivo estiver entre os símbolos (< e >) — usado para os *arquivos de cabeçalho da biblioteca padrão* — a procura é realizada de um modo que varia conforme a implementação, normalmente através de diretórios predefinidos.

A diretiva **#include** é usada normalmente para incluir arquivos de cabeçalho da biblioteca padrão como **stdio**.he **stdlib.h** (vejaaFig. 5.6). A diretiva **#include** também é usada com programas que consistem em vários arquivos-fonte que devem ser compilados em conjunto. Freqüentemente é criado um *arquivo de cabeçalho* contendo declarações comuns a todos os arquivos de programas onde tal arquivo é incluído. Exemplos de tais declarações são as declarações de estruturas e uniões, enumerações e protótipos de funções.

Em UNIX, os arquivos de programas são compilados usando o *comando **cc.*** Por exemplo, para compilar e linkar **main.ce square.c** digite o comando

cc main.c square.c

no prompt do UNIX. Isso produz o arquivo executável **a**. out. Veja os manuais de referência de seu compilador para obter mais informações sobre compilação, linkagem e execução de programas.

13.3 A Diretiva #define do Pré-processador: Constantes Simbólicas

A *diretiva* ***#define*** cria *constantes simbólicas* — constantes representadas por símbolos — e *macros* — operações definidas como símbolos. O formato da diretiva **#define** é

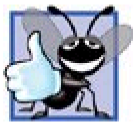
#define *identificador   texto de substituição*

Quando essa linha aparece em um arquivo, todas as ocorrências subseqüentes de *identificador* serão substituídas automaticamente por *texto de substituição* antes de o programa ser compilado. Por exemplo,

#define PI 3.14159

substitui todas as ocorrências subseqüentes da constante simbólica **PI** pela constante numérica **3.14159.**

As constantes simbólicas permitem que o programador crie um nome para a constante e use o nome ao longo de todo o programa. Se a constante precisar ser modificada no programa, ela pode ser modificada dentro da diretiva **#define** — e quando o programa voltar a ser compilado, todas as ocorrências da constante no programa serão modificadas automaticamente. Observação: *Tudo à direita do nome da constante simbólica substitui a constante simbólica.* Por exemplo, **#define PI = 3.14159** faz com que o pré-processador substitua todas as ocorrências de **PI** por = **3.14159.** Isso é a causa de muitos erros lógicos e de sintaxe sutis. Redefinir a constante simbólica com um novo valor também configura um erro.



Boa prática de programação 13.1

Usar nomes significativos para as constantes simbólicas ajuda a tornar os programas mais auto-explicativos.

13.4 A Diretiva #define do Pré-processador: Macros

Uma *macro* é uma operação definida em uma diretiva **#def ine** do pré-processador. Da mesma forma que as constantes simbólicas, o *identificador da macro* é substituído no programa por um *texto de substituição* antes de o programa ser compilado. As macros podem ser definidas com ou sem *argumentos.*

Uma macro sem argumentos é processada como uma constante simbólica. Em uma macro com argumentos, esses são substituídos no texto de substituição, então a macro é *expandida* — i.e., o texto de substituição substitui o identificador e a lista de argumentos no programa.

Considere a seguinte definição de macro com um argumento para calcular a área de um círculo:

#define AREA_CIRCULO(x) (PI * (x) * (x))

Sempre que **AREA_CIRCULO (x)** aparecer no arquivo, o valor de **x** substitui **x** no texto de substituição, a constante simbólica **PI** é substituída por seu valor (definido anteriormente) e a macro é expandida no programa. Por exemplo, a instrução

area = AREA_CIRCULO(4);

é expandida em

area = (3.14159 * (4) * (4));

Como a expressão consiste apenas em constantes, durante a compilação, o valor da expressão é calculado e atribuído à variável **area.** Os parênteses em torno de cada **x** no texto de substituição impõem a ordem adequada de cálculo quando o argumento da macro for uma expressão. Por exemplo, a instrução

area = AREA_CIRCULO(c + 2);

é expandida em

area = (3.14159 * (c + 2) * (c + 2));

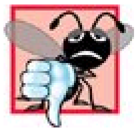
que é calculada corretamente porque os parênteses impõem a ordem adequada de cálculo. Se os parênteses fossem omitidos, a expansão da macro seria

area = 3.14159 *c+2*c+2;

que é calculada erradamente como

area = (3.14159 * c) + (2 * c) + 2;

por causa das regras de precedência de operadores.



Erro comun de programação 13.1

Esquecer-se de colocar os argumentos da macro entre parênteses no texto de substituição.

A macro **AREA_CIRCULO** poderia ser definida como uma função. A função **areaCirculo**

```
double areaCirculo(double x)   {
 return 3.14159 * x * x;
 }
```

realiza os mesmos cálculos que a macro **AREA_CIRCULO,** mas o overhead de uma chamada de função é associado com a função **areaCirculo.** As vantagens da macro **AREA_CIRCULO** são que as macros inserem código diretamente no programa — evitando o overhead das funções — e o programa permanece legível porque o cálculo de **AREA_CIRCULO** é definido separadamente e recebe um nome significativo. Uma desvantagem é que seu argumento é calculado duas vezes.



Dica de desempenho 13.1

Algumas vezes as macros podem ser usadas para substituir uma chamada de função por código em linha antes da compilação. Isso elimina o overhead de uma chamada de função.

A seguir está uma definição de macro com 2 argumentos para a área de um retângulo:

```
#define AREA_RETANGULO(x,  y)   (   (x)   *   (y) )
```

Sempre que **AREA_RETANGULO (x, y)** aparecer no programa, os valores de **x** e **y** são substituídos no texto de substituição da macro e a macro é expandida no lugar de seu nome. Por exemplo, a instrução

```
areaRet = AREA_RETANGULO(a + 4, b + 7);
```

é expandida em

```
areaRet =  (  (a + 4)  *  (b + 7)   );
```

O valor da expressão é calculado e atribuído à variável **areaRet.**

O texto de substituição de uma macro ou constante simbólica é normalmente qualquer texto na linha depois do identificador na diretiva **#define.** Se o texto de substituição de uma macro ou constante simbólica for maior do que o restante da linha, deve ser colocada uma barra invertida (backslash, \) no final da linha, para indicar que o texto de substituição continua na próxima linha.

As constantes simbólicas e macros podem ser eliminadas por meio da *diretiva de pré-processador* ***#undef.*** A diretiva **ftundef** "anula a definição" de um nome de constante simbólica ou macro. O *escopo* de uma constante simbólica ou macro compreende desde sua definição até essa ser desfeita com **#undef,** ou até o final do arquivo. Uma vez desfeita a definição, um nome pode ser redefinido com **#define.**

Algumas vezes, as funções na biblioteca padrão são definidas como macros baseadas em outras funções da biblioteca. Uma macro definida normalmente no arquivo de cabeçalho **stdio. h** é

#define getchar() getc(stdin)

A definição de macro de **getchar** usa a função **getc** para obter um caractere do dispositivo padrão de entrada. A função **putchar** do cabeçalho **stdio. h** e as funções de manipulação de caracteres do cabeçalho **ctype. h** também são implementadas freqüentemente como macros. Observe que expressões com efeitos secundários (i.e., os valores das variáveis são modificados) não devem ser passadas a uma macro porque os argumentos da macro podem ser calculados mais de uma vez.

13.5 Compilação Condicional

A *compilação condicional* permite que o programador controle a execução das diretivas do pré-processador e a compilação do código do programa. Cada uma das diretivas condicionais do pré-processador calcula uma expressão constante inteira. Expressões de conversão, expressões **sizeof** e constantes de enumeração não podem ser calculadas em diretivas do pré-processador.

O conceito fundamental do pré-processador condicional é muito parecido com a estrutura de seleção **if**. Considere o seguinte código de pré-processador:

#if !defined(NULL) ftdefine NULL 0 #endif

Essas diretivas determinam se **NULL** está definido. A expressão **def ined (NULL)** fornece o valor **1** se **NULL** for definido; **0** em caso contrário. Se o resultado for **0, !defined (NULL)** fornece o valor **1** e **NULL** é definido. Caso contrário, adiretiva **#define** é ignorada. Todo bloco **#if** termina com **#endif.**

As diretivas **#ifdef e *#ifndef*** são abreviações de **#if defined** *(nome)* e**#if !defined** *(nome).*

Um bloco condicional de pré-processador com várias partes pode ser testado usando as diretivas **#elif** (o equivalente a **else if** em uma estrutura **if**) e **#else** (o equivalente a **else** em uma estrutura **if).**

Durante o desenvolvimento do programa, freqüentemente os programadores acham útil "comentar" grandes partes de código para evitar que ele seja compilado. Se o código possuir comentários, / * e * / não podem ser usados para realizar essa tarefa. Em vez disso, o programador pode usar o seguinte bloco de pré-processador

#if 0

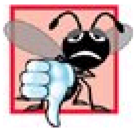
código cuja compilação se quer evitar **#endif**

Para permitir que o código seja compilado, o **0** no bloco anterior é substituído por **1.**

A compilação condicional é usada normalmente como uma ajuda na depuração (debugging) do programa. Muitas implementações do C fornecem *depuradores (debuggers).* Entretanto, freqüentemente os depuradores são difíceis de usar e entender, e por isso raramente são usados por alunos de um curso básico de programação. Em vez disso, são usadas instruções **printf** para imprimir o valor das variáveis e para confirmar o fluxo do controle. Essas instruções **printf** podem ser colocadas entre as diretivas condicionais do pré-processador para que sejam compiladas apenas enquanto o processo de depuração não for concluído. Por exemplo,

#ifdef DEBUG
printf("Variável x = %d\n", x); #endif

faz com que uma instrução **printf** seja compilada no programa se a constante simbólica **DEBUG** tiver ido definida **(#def ine DEBUG)** antes da diretiva **#ifdef DEBUG.** Quando a depuração estiver concluída, a diretiva **#def ine** é removida do arquivo-fonte e as instruções **printf,** inseridas apenas com o objetivo de auxiliar na depuração, são ignoradas durante a compilação. Em programas grandes, pode ser desejável definir várias constantes simbólicas diferentes que controlam a compilação condicional em seções separadas do arquivo-fonte.



Erro comun de programação 13.2

*Inserir instruções **printf** compiladas condicionalmente com a finalidade de auxiliarna depuração em locais onde a linguagem C espera uma instrução simples. Nesse caso, a instrução compilada condicionalmente deve ser colocada em uma instrução composta. Dessa forma, quando o programa for compilado com instruções de depuração, o fluxo de controle do programa não é alterado.*

13.6 As Diretivas #error e #pragma do Pré-processador

A *diretiva* ***#error***

#error *segmentos*

imprime uma mensagem, dependente da implementação, que inclui os *segmentos (tokens)* especificados na diretiva. Os segmentos especificados (tokens) são seqüências de caracteres separados por espaços. Por exemplo,

#error 1 — Out of range error

contém 6 segmentos. No Borland C++ para PCs, por exemplo, quando uma diretiva **Ierror** é processada, os segmentos da mensagem na diretiva são exibidos como uma mensagem de erro, o pré-processamento é interrompido e o programa não é compilado.

A *diretiva* ***#pragma*** **#pragma** *segmento*

causa uma ação definida na implementação. Um pragma não reconhecido pela implementação é ignorado. O Borland C++, por exemplo, reconhece vários pragmas que permitem ao programador tirar o máximo proveito da implementação. Para obter mais informações sobre **#error** e **#pragma,** veja a documentação de sua implementação da linguagem C.

13.7 Os Operadores # e

Os operadores de pré-processador # e ## estão disponíveis apenas em ANSI C. O operador # faz com que o segmento de um texto de substituição seja convertido em uma string entre aspas. Considere a seguinte definição de macro:

#define HELLO(x) printf("Hello,   " #x "An");

Quando **HELLO (Paulo)** aparecer em um arquivo de programa, isso é expandido em

printf("Hello,"   "Paulo" "\n");

A string **"Paulo"** substitui **#x** no texto de substituição. As strings separadas por espaços em branco são concatenadas durante o pré-processamento, portanto a expressão anterior é equivalente a

printf("Hello, PauloXn");

Observe que o operador # deve ser usado em uma macro com argumentos porque o operando de # se refere a um argumento da macro.

O operador ## concatena dois segmentos. Considere a seguinte definição de macro:

#define TOKENCONCAT(x, y) x ## y

Quando **TOKENCONCAT** aparecei" no programa, seus argumentos são concatenados e usados para substituir a macro. Por exemplo, **TOKENCONCAT (O, K)** é substituído por **OK** no programa. O operador ## deve ter dois operandos.

13.8 Números de Linhas

A *diretiva de pré-processador* ***#line*** faz com que as linhas de código-fonte subseqüentes sejam renumeradas, iniciando com o valor constante inteiro especificado. A diretiva

#line 100

começa a numeração de linhas a partir de **10 0** iniciando com a próxima linha de código-fonte. Um nome de arquivo pode ser incluído na diretiva **#line.** A diretiva

#line 100 "arquivol.c"

indica que as linhas são numeradas a partir de **100,** iniciando com a próxima linha do código-fonte e que o nome do arquivo para receber qualquer mensagem do compilador é " **arquivo1. c**". A diretiva é usada normalmente para ajudar a tornar mais significativas as mensagens produzidas por erros de sintaxe e avisos do compilador. Os números de linhas não aparecem no arquivo-fonte.

Constante simbólica	Explicação
__LINE__	O número da linha de código-fonte atual (uma constante inteira)
__FILE__	O nome cedido ao arquivo-fonte (uma string)
__DATE__	A data na qual o arquivo-fonte é compilado (uma string da forma “Mmm dd aaaa” como “Jan 21 1999”).
__TIME__	A hora na qual o arquivo-fonte é compilado (uma string literal da forma **"hh:mm:ss").**
__STDC__	A constante inteira 1. Isso tem a finalidade de indicar que a implementação está de acordo com o ANSI.

Fig. 13.1 As constantes simbólicas predefinidas.

13.9 Constantes Simbólicas Predefinidas

Há cinco *constantes simbólicas predefinidas* (Fig. 13.1). Os identificadores de cada uma das constantes simbólicas predefinidas utilizam *dois* caracteres sublinhados, um no início do identificador e outro no fim. Esses identificadores e o identificador **def ined** (usado na Seção 13.5) não podem ser usados em diretivas **#def ine** ou **#undef.**

13.10 Assertivas

A *macro assert* — definida no arquivo de cabeçalho **assert. h —** verifica o valor de uma expressão. Se o valor da expressão for **0** (falso), então **assert** imprime uma mensagem de erro e chama a função *abort* (da biblioteca geral de utilitários — **stdlib. h)** para encerrar a execução do programa. Isso é uma ferramenta útil de depuração para verificar se uma variável está com o valor correto. Por exemplo, suponha que a variável **x** nunca deve ser maior que **10** em um programa. Pode ser usada uma assertiva para verificar o valor de **x** e imprimir uma mensagem de erro se o valor de **x** estiver incorreto. A instrução poderia ser:

```
assert(x <= 10);
```

Se **x** for maior que **10** quando a instrução anterior for encontrada em um programa, será impressa uma mensagem de erro contendo o número da linha e o nome do arquivo, e o programa será encerrado. O programador pode então examinar essa área do código para encontrar o erro. Se a constante simbólica **NDEBUG** estiver definida, as assertivas subseqüentes serão ignoradas. Assim, quando as assertivas não forem mais necessárias, a linha

```
#define NDEBUG
```

é inserida no arquivo de programa em vez de as assertivas serem apagadas manualmente.

Resumo

• Todas as diretivas do pré-processador começam com #.

• Apenas os caracteres de espaço em branco podem aparecer antes de uma diretiva de pré-processador em uma linha.

• A diretiva #**include** inclui uma cópia do arquivo especificado. Se o nome do arquivo estiver entre aspas, o pré-processador começa a procurar pelo arquivo a ser incluído no mesmo diretório do arquivo que está sendo compilado. Se o nome do arquivo estiver entre os sinais < e >, a procura é realizada de uma forma que varia conforme a implementação.

• A diretiva de pré-processador **#define** é usada para criar constantes simbólicas e macros.

• A constante simbólica é um nome de uma constante.

• Uma macro é uma operação definida em uma diretiva de pré-processador **#define**. As macros podem ser definidas com ou sem argumentos.

• O texto de substituição de uma macro ou constante simbólica é qualquer texto que resta na linha depois do identificador na diretiva **#def ine.** Se o texto de substituição de uma macro ou constante simbólica for maior que o restante da linha, é colocada uma barra invertida (\) no final da linha, indicando que o texto de substituição continua na próxima linha.

• As constantes simbólicas e macros podem ser eliminadas por meio da diretiva de pré-processador **#undef**. A diretiva **#undef** "anula a definição" do nome da constante simbólica ou da macro.

• O escopo de uma constante simbólica ou macro compreende desde sua definição até essa ser anulada com **#undef** ou até o final do arquivo.

• A compilação condicional permite que o programador controle a execução das diretivas do pré-processador e a compilação do código do programa.

• As diretivas condicionais do pré-processador calculam expressões constantes inteiras. Expressões de conversão, expressões **sizeof** e constantes de enumeração não podem ser calculadas nas diretivas do pré-processador.

• Todo bloco **#if** termina com **#endif.**

• As diretivas **#ifdef** e **#ifndef** são fornecidas como abreviações de **#if defined** *(nome)* e **#if !defined***(nome).*

• Um bloco condicional do pré-processador, em várias partes, pode ser testado usando as diretivas **#elif** (o equivalente a **else if** em uma estrutura **if**) e **#else** (o equivalente a **else** em uma estrutura **if).**

• A diretiva **#error** imprime uma mensagem que varia conforme a implementação e que inclui os segmentos de texto especificados na diretiva.

• A diretiva **#pragma** origina uma ação definida na implementação. Se o pragma não for reconhecido pela implementação, será ignorado.

• O operador # faz com que um segmento de texto de substituição seja convertido em uma string colocada entre aspas. O operador # deve ser usado em uma macro com argumentos porque o operando de # deve ser um argumento daquela macro.

• O operador ## concatena dois segmentos de texto. O operador ## deve ter dois operandos.

• A diretiva de pré-processador **#line** faz com que as linhas subseqüentes de código-fonte sejam renumeradas, iniciando com o valor constante inteiro especificado.

• Há cinco constantes simbólicas predefinidas. A constante **__LINE__** é o número da linha do código-fonte atual (um inteiro). A constante**_FILE_**é o nome cedido ao arquivo (uma string). A constante **_DATE_**é a data na qual o arquivo-fonte é

compilado (uma string). A constante **__TIME__** é a hora na qual o arquivo-fonte é compilado (uma string). A constante **__STDC__** é **1;** ela tem a finalidade de indicar que a implementação está de acordo com o ANSI. Observe que cada uma das constantes simbólicas predefinidas começa e termina com dois caracteres sublinhados.

• A macro **assert** — definida no arquivo de cabeçalho **assert. h** — verifica o valor de uma expressão. Se o valor da expressão for **0** (falso), **assert** imprime uma mensagem de erro e chama a função **abort** para terminar a execução do programa.

Terminologia

#define **#elif** **#else** **#endif** **#error** **#if** **#ifdef** **#ifndef** **#include <nome-do-arquivo>** **#include "nome-do-arquivo"** **#line** **#pragma** **#undef** **_DATE_** **FILE** **_LINE_** **_STDC_** **_TIME_** **a.out** em UNIX **abort** argumento arquivo de cabeçalho arquivos de cabeçalho da biblioteca padrão **assert** **assert.h**	comando cc em UNIX compilação condicional constante simbólica constantes simbólicas predefinidas caractere de continuação \ (barra invertida ou backslash) debugger depurador diretiva de pré-processamento escopo de uma constante simbólica ou macro execução condicional das diretivas do pré-processador expandir uma macro macro macro com argumentos operador # do pré-processador para conversão de strings operador ## do pré-processador para concatenação pré-processador C **stdio.h** **stdlib.h** texto de substituição

Erros Comuns de Programação

13.1 Esquecer-se de colocar os argumentos da macro entre parênteses no texto de substituição.

13.2 Inserir instruções **printf** compiladas condicionalmente com a finalidade de auxiliar na depuração em locais onde a linguagem C espera uma instrução simples. Nesse caso, a instrução compilada condicionalmente deve ser colocada em uma instrução composta. Dessa forma, quando o programa for compilado com instruções de depuração, o fluxo de controle do programa não é alterado.

Prática Recomendável de Programação

13.1 Usar nomes significativos para as constantes simbólicas ajuda a tornar os programas mais auto-explicativos.

Dica de Performance

13.1 Algumas vezes as macros podem ser usadas para substituir uma chamada de função por código em linha antes da compilação. Isso elimina o overhead de uma chamada de função.

Exercícios de Revisão

13.1 Preencha as lacunas de cada uma das seguintes sentenças:

a) Todas as diretivas do pré-processador devem começar com_.

b) O bloco de compilação condicional pode ser ampliado para testar vários casos usando as diretivas _e_.

c) A diretiva_cria macros e constantes simbólicas.

d) Apenas caracteres_podem aparecer antes de uma diretiva de pré-processador em uma linha.

e) A diretiva_anula os nomes de constantes simbólicas e macros.

f) As diretivas _e _ são fornecidas como notação abreviada para **#if defined**(nome) e **#if !defined***(nome).*

g) _permite ao programador controlar a execução das diretivas do pré-processador e a compilação do código do programa.

h) A macro_imprime uma mensagem e encerra a execução do programa se o valor da expressão que a macro calcula é 0.

i) A diretiva_insere um arquivo em outro arquivo.

j) O operador_concatena dois argumentos.

k) O operador_converte seu operando em uma string.

1) O caractere_indica que o texto de substituição de uma constante simbólica ou macro continua na próxima linha.

m) A diretiva_ faz com que as linhas do código-fonte sejam numeradas a partir do valor indicado começando na próxima linha do código-fonte.

13.2 Escreva um programa para imprimir os valores das constantes simbólicas predefinidas listadas na Fig. 13.1.

13.3 Escreva uma diretiva do pré-processador para realizar cada um dos pedidos a seguir:

a) Defina a constante simbólica **YES** com o valor **1.**

b) Defina a constante simbólica **NO** com o valor **0.**

c) Inclua o arquivo de cabeçalho **common. h.** O cabeçalho se encontra no mesmo diretório do arquivo que está sendo compilado.

d) Renumere as linhas restantes no arquivo, começando com o número de linha **300 0.**

e) Se a constante simbólica **VERDADE** estiver definida, anule sua definição e redefina-a como **1.** Não use **#ifdef.**

f) Se a constante simbólica **VERDADE** estiver definida, anule sua definição e redefina-a como **1.** Use a diretiva de pré-processador **#ifdef.**

g) Se a constante simbólica **VERDADE** não for igual a **0,** defina como **0** a constante simbólica **FALSO.** Caso contrário, defina **FALSO** como **1.**

h) Defina a macro **VOLUME_CUBO** que calcula o volume de um cubo. A macro utiliza um argumento.

Respostas dos Exercícios de Revisão

13.1 a) #. b) **#elif, #else.** c) **#def ine.** d) de espaço em branco, e) **#undef**. f) **#ifdef, ifndef. g) A** compilação cond

13.2

```
/* Imprime os valores das macros predefinidas */
#include <stdio.h>
main( ){

        printf("_LINE_     = %d\n", _LINE_);
        printf("_FILE_     = %s\n", _FILE_);
        printf ("_DATE_   = %s\n", _DATE_) ;
        printf("_TIME_     = %s\n", _TIME_);
        printf("_STDC_     = %d\n", _STDC_);
}
```

```
__LINE__ =  5
__FILE__ = macros.c
__DATE__ = Sep 08 1993
__TIME__ =  10:23:47
__STDC__ = 1
```

13.3

a) **#define YES 1**
b) **#define NO 0**
c) **#include "common.h"**
d) **#line 3000**
e) **#if defined(VERDADE)**
#undef VERDADE #define VERDADE 1 #endif
f) **#ifdef VERDADE**
#undef VERDADE #define VERDADE 1 #endif
g) **#if VERDADE**
#define FALSO 0 #else
#define FALSO 1 #endif

h) **#define VOLUME_CUBO(x)   (x) * (x) * (x)**

Exercícios

13.4 Escreva um programa que defina uma macro com um argumento para calcular o volume de uma esfera. O programa deve calcular o volume de esferas com raios de 1 a 10 e imprimir os resultados em um formato de tabela. A fórmula para o volume de uma esfera é:

$$(4/3) * PI * r^3$$

onde PI é **3.14159.**

13.5 Escreva um programa que produza a seguinte saída:

A soma de x e y vale 13

O programa deve definir a macro **SOMA** com dois argumentos, **x** e **y,** e usar **SOMA** para produzir a saída.

13.6 Escreva um programa que use a macro **MINIMUM2** para determinar o menor de dois valores numéricos. Entre com os valores a partir do teclado.

13.7 Escreva um programa que use a macro **MINIMUM3** para determinar o menor de três valores numéricos. A macro **MINIMUM3** deve usar a macro **MINIMÜM2** definida no Exercício 13.6 para determinar o menor número. Entre com os valores a partir do teclado.

13.8 Escreva um programa que use a macro **IMPRIMIR** para imprimir o valor de uma string.

13.9 Escreva um programa que use a macro **IMPRIME ARRAY** para imprimir um array de inteiros. A macro deve receber como argumentos o array e o seu número de elementos.

13.10 Escreva um programa que use a macro **SOMAARRAY** para somar os valores de um array numérico. A macro deve receber como argumentos o array e o seu número de elementos.

Apêndice A

Biblioteca Padrão

A.1 Erros <errno.h>

EDOM
ERANGE

Essas geram expressões constantes inteiras com valores diferentes de zero, adequadas para uso em diretivas de pré-processamento **#if.**

errno

Um valor do tipo **int** que é definido com um número positivo de erro por várias funções da biblioteca. O valor de **errno** é igual a zero no início do programa, mas nunca é definido como zero por qualquer função da biblioteca. Um programa que usa **errno** para verificação de erros deve defini-la como zero antes da chamada da função da biblioteca e verificá-la antes de uma chamada a uma função subseqüente da biblioteca. Uma função da biblioteca pode salvar o valor de **errno** na entrada e então defini-la como zero, contanto que o valor original seja restaurado se o valor de **errno** ainda for zero imediatamente antes de retornar. O valor de **errno** pode ser definido pela chamada de uma função da biblioteca com um valor diferente de zero se houver ou não um erro, já que o uso de **errno** não é documentado na descrição da função no padrão.

A.2 Definições Comuns <stddef.h>

NULL

Um ponteiro constante nulo definido na implementação.

offsetof *(tipo, designador de membro)*

Gera uma expressão constante inteira do tipo **size_t,** cujo valor é a diferença em bytes de um membro da estrutura (designado por *designador de membro)* desde o início de seu tipo de estrutura (designado por *tipo). O designador de membro* deve ser tal que dado

static *tipo* ***t;***

então a expressão &(t.*designador de membro)* resulta em uma constante de endereço. (Se o membro especiicado for um campo de bit, o comportamento fica indefinido.)

ptrdiff_t

O tipo inteiro com sinal do resultado da subtração de dois ponteiros.

size_t

O tipo inteiro com sinal do resultado do operador **sizeof.**

wchar_t

Um tipo inteiro cujo intervalo de valores pode representar códigos distintos de todos os membros do maior conjunto de caracteres estendido especificado entre os locais suportados; o caractere nulo (null) deve ter o valor de código zero e cada membro do conjunto básico de caracteres deve ter um valor de código igual a seu valor quando usado como caractere isolado em uma constante inteira de caracteres.

A.3 Diagnósticos <assert.h>

void assert(int valor);

A macro **assert** insere diagnósticos em programas. Quando ela é executada, se **valor** for falso, a macro **assert** escreve informações sobre a chamada específica que falhou (incluindo o texto do argumento, o nome do arquivo-fonte e o número de sua linha — os últimos são os valores respectivos das macros de pré-processamento **_FILE_** e **_LINE_)** no arquivo padrão de erros em um formato que varia de acordo com as definições de implementação. A mensagem escrita pode ser da forma

Assertion failed: *valor*, **file xyz, line** *nnn*

A macro **assert** chama então a função **abort.** Se a diretiva de pré-processador

#define NDEBUG

aparecer no arquivo-fonte onde **assert.h** está incluído, todas as assertivas do arquivo são ignoradas.

A.4 Manipulação de Caracteres <ctype.h>

As funções desta seção retomam valor diferente de zero (verdadeiro) se e somente se o valor do argumento c obedecer a isso na descrição da função.

int isalnum(int c);
Verifica a existência de qualquer caractere para o qual **isalpha** ou **isdigit** é verdadeiro.

int isalpha(int c);
Verifica a existência de qualquer caractere para o qual **isupper** ou **islower** é verdadeiro.

int iscntrl(int c);
Verifica a existência de qualquer caractere de controle.

int isdigit(int c);
Verifica a existência de qualquer caractere que seja dígito decimal.

int isgraph(int c);
Verifica a existência de qualquer caractere imprimivel exceto espaço (' ').

int islower(int c);
Verifica a existência de qualquer caractere que seja uma letra minúscula.

int isprint(int c);
Verifica a existência de qualquer caractere imprimivel incluindo espaço (` `).

int ispunct(int c);
Verifica a existência de qualquer caractere imprimivel que não seja espaço nem um caractere para o qual isalnum seja verdadeiro.

int isspace(int c);
Verifica a existência de qualquer caractere que seja um caractere padrão de espaço em branco. Os caracteres padrões de espaço em branco: espaço (' '), alimentação de formulário (' \ **f '),** nova linha (' **\n'),** carriage return (' **\r'),** tabulação horizontal (' **\t') e** tabulação vertical (' **\v').**

int isupper(int c);
Verifica a existência de qualquer caractere que seja uma letra maiúscula.

int lsxdigit(int c);
Verifica a existência de qualquer caractere que seja dígito hexadecimal.

int tolower(int c);
Converte uma letra maiúscula na letra minúscula correspondente. Se o argumento for um caractere para o qual **isupper** é verdadeiro e houver um caractere correspondente para o qual **islower** é verdadeiro, a função **tolower** retorna o caractere correspondente; caso contrário, o argumento retorna inalterado.

int toupper(int c);

Converte uma letra minúscula na letra maiúscula correspondente. Se o argumento for um caractere para o qual **islower** é verdadeiro e houver um caractere correspondente para o qual **isupper** é verdadeiro, a função **toupper** retorna o caractere correspondente; caso contrário, o argumento retorna inalterado.

A.5 Localização <locale.h>

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Essas geram expressões constantes inteiras com valores distintos, adequadas para uso como primeiro argumento da função **setlocale.**

NULL

Uma constante nula de ponteiro definida na implementação.

struct lconv

Contém membros relacionados com a formatação de valores numéricos. A estrutura deve conter pelo menos os seguintes membros, em qualquer ordem. No local **"C",** os membros devem ter os valores especificados nos comentários.

```
char *decimal_point; /*  "." */
char *thousands-sep; /*  "" */
char *grouping; /*  "" */
char *int_curr_symbol; /*  "" */
char *currency_symbol; /*  "" */
char *mon_decimal_point; /*  "" */
char *mon_thousands_sep; /*  "" */
char *mon_grouping; /*  "" */
char *positive_sign; /*  "" */
char *negative_sign; /*  "" */
char int_frac_digits; /* CHAR_MAX */
char frac_digits; /* CHAR_MAX */
char p_cs_precedes; /* CHAR_MAX */
char p_sep_by_space; /* CHAR_MAX */
char n_cs_precedes; /* CHAR_MAX */
char n_sep_by_space; /* CHAR_MAX */
char p_sign_posn; /* CHAR_MAX */
char n_sign_posn; /* CHAR_MAX */
```

char *setlocale(int category, const char *locale);

A função **setlocale** seleciona a parte apropriada do local do programa de acordo com o especificado nos argumentos **category** e **locale.** A função **setlocale** pode ser usada para modificar ou verificar o local atual de todo o programa ou de suas partes. O valor **LC_ALL** para **category** fornece um nome a todo o local do programa. **LC_COLLATE** afeta o comportamento das funções **strcoll** e **strxfrm. LC_CTYPE** afeta o comportamento das funções de manipulação de caracteres e das funções de vários bytes. **LC_MONETARY** afeta as informações de formatação de valores monetários retornadas pela função **localeconv. LC_NUMERIC** afeta o caractere de ponto decimal para as funções de entrada/saída formatadas, para as funções de conversão de strings e para as informações de formatação de valores não-monetários retomadas por **localeconv. LC_TIME** afeta o comportamento de **strftime.**

O valor **"C"** para **locale** especifica o ambiente mínimo para tradução do C; um valor "" para **locale** especifica o ambiente nativo definido na implementação. Podem ser passadas para **setlocale** outras strings definidas na implementação. No início da execução do programa, o equivalente a

setlocale(LC_ALL, "C");

é executado. Se for indicado um ponteiro para uma string para **locale** e a seleção puder ser realizada, a função **setlocale** retorna um ponteiro para uma string associada com a categoria **(category)** especificada para o novo local. Se a seleção não puder ser realizada, a função **setlocale** retorna um ponteiro nulo e o local do programa não é modificado.

Um ponteiro nulo para **locale** faz com que a função **setlocale** retorne um ponteiro para a string associada com a categoria **(category)** do local atual do programa; o local do programa não é modificado.

O ponteiro para string retornado pela função **setlocale** é tal que a chamada subseqüente com o valor daquela string e sua categoria associada recuperarão aquela parte do local do programa. A string apontada deve ser modificada pelo programa, mas pode ser sobrescrita por uma chamada subseqüente à função **setlocale.**

struct lconv *localeconv(void);

A função **localeconv** define os componentes de um objeto com o tipo **struct lconv** com valores apropriados para a formatação de quantias numéricas (monetárias e outras) de acordo com as regras do local atual.

Os membros da estrutura com tipo **char** * são ponteiros para strings, sendo que qualquer um deles (exceto **decimal_point)** pode apontar para " " para indicar que o valor não está disponível no local atual ou que tem comprimento zero. Os membros com tipo **char** são números não-negativos e qualquer um deles pode ser **CHAR_MAX** para indicar que o valor não está disponível no local atual. Os membros indicam o seguinte:

char *decimal point
O caractere de ponto decimal usado para formatar valores não-monetários.

char *thousands_sep
O caractere usado para separar grupos de dígitos antes do caractere de ponto decimal em valores não-monetários formatados.

char *grouping
Uma string cujos elementos indicam o tamanho de cada grupo de dígitos em valores não-monetários formatados.

char *int_curr_symbol
O símbolo internacional de valores monetários aplicável ao local atual. Os três primeiros caracteres contêm o símbolo monetário alfabético internacional de acordo com o especificado na ISO 4217:1987. O quarto caractere (imediatamente antes do

caractere nulo) é o caractere usado para separar o símbolo monetário internacional do valor monetário.

char *currency_symbol
O símbolo monetário local aplicável ao local atual.

char *mon_decimal_point
O ponto decimal usado para formatar quantidades monetárias.

char *mon_thousands_sep
O separador de grupos de dígitos antes do ponto decimal em valores monetários formatados.

char *mon_grouping
Uma string cujos elementos indicam o tamanho de cada grupo de dígitos em valores monetários formatados.

char *positive_sign
Uma string usada para indicar um valor monetário formatado positivo.

char *negative_sign
Uma string usada para indicar um valor monetário formatado negativo.

char int_frac_digits
O número de dígitos fracionários (aqueles após o ponto decimal) a serem exibidos em um valor monetário formatado internacionalmente.

char frac_digits
O número de dígitos fracionários (aqueles após o ponto decimal) a serem exibidos em um valor monetário formatado.

char p_cs_precedes
Ao ser definido como 1 ou 0, **currency_symbol** precede ou sucede, respectivamente, o valor de uma quantia monetária formatada não-negativa.

char p_sep_by_space
Definido como 1 ou 0, **currency__symbol** é separado ou não, respectivamente, por um espaço de um valor de uma quantia monetária formatada não-negativa.

char n_cs_precedes
Definido como 1 ou 0, **currency_symbol** precede ou sucede, respectivamente, o valor de uma quantia monetária formatada negativa.

char n_sep_by_space
Definido como 1 ou 0, **currency_symbol** é separado ou não, respectivamente, por um espaço de um valor de uma quantia monetária formatada negativa.

char p_sign_posn
Definido com um valor que indica o posicionamento de **positive_sign** para uma quantia monetária formatada não-negativa.

char n_sign_posn
Definido com um valor que indica o posicionamento de **negative_sign** para uma quantia monetária formatada negativa.

Os elementos de **grouping** e **mon_grouping** são interpretados de acordo com o seguinte:
CHAR_MAX Não é realizado nenhum agrupamento.
0 elemento anterior deve ser usado repetidamente para o restante dos dígitos.
outro O valor inteiro é o número de dígitos que constitui o grupo atual. O próximo elemento é examinado para determinar o tamanho do próximo grupo de dígitos antes do grupo atual.

Os valores de **p_sign_posn** e **n_sign_posn** são interpretados de acordo com o seguinte:

0 São colocados parênteses em torno da quantia e de **currency_symbol.**
1 A string de sinal antecede a quantia e **currency_symbol.**
2 A string de sinal sucede a quantia e **currency_symbol.**
3 A string de sinal antecede imediatamente **currency_symbol.**
4 A string de sinal sucede imediatamente **currency_symbol.**

A função **localeconv** retorna um ponteiro para o objeto preenchido. A estrutura apontada pelo valor de retorno não deve ser modificada pelo programa, mas pode ser sobrescrita por uma chamada subseqüente à função **localeconv.** Além disso, as chamadas à função **setlocale** com categorias **LC_ALL, LC_MONETARY** ou **LC_NUMERIC** podem sobrescrever o conteúdo da estrutura.

A.6 Matemática <math. h>

HUGE_VAL

Uma constante simbólica que representa uma expressão positiva **double.**

double acos(double x);

Calcula o valor principal do arco cosseno de **x.** Ocorre um erro de domínio para argumentos que não estejam no intervalo [— 1, +1]. A função **acos** retorna o arco cosseno no intervalo [0, **Pi]** radianos.

double asin(double x);

Calcula o valor principal do arco seno de **x.** Ocorre um erro de domínio para argumentos que não estejam no intervalo [-— 1, +1]. A função **asin** retorna o arco seno no intervalo **[—Pi**/2, + **Pi**/2] radianos.

double atan(double x);

Calcula o valor principal do arco tangente de **x.** A função **atan** retorna o arco tangente no intervalo [-**Pi**/2, **Pi**/2] radianos.

double atan2(double y, double x);

A função **atan2** calcula o valor principal do arco tangente de **y/x,** usando os sinais de ambos os argumentos para determinar o quadrante do valor de retorno. Pode ocorrer um erro de domínio se ambos os argumentos forem zero. A função **atan2** retorna o arco tangente de **y/x** no intervalo **[—Pi, +Pi]** radianos.

double cos(double x);

Calcula o cosseno de **x** (medido em radianos).

double sin(double x) ;

Calcula o seno de **x** (medido em radianos).

double tan(double x) ;

Calcula a tangente de **x** (medida em radianos).

double cosh(double x);

Calcula o cosseno hiperbólico de **x.** Ocorre um erro de dimensão se a magnitude de **x** for muito grande.

double sinh(double x);

Calcula o seno hiperbólico de **x.** Ocorre um erro de dimensão se a magnitude de **x** for muito grande.

double tanh(double x);

A função **tanh** calcula a tangente hiperbólica de **x.**

double exp(double x);

Calcula a função exponencial de **x.** Ocorre um erro de dimensão se a magnitude de **x** for muito grande.

double frexp(double valor, int *exp);

Divide o número de ponto flutuante em uma fração normalizada e uma potência inteira de 2. Ela armazena o inteiro no objeto **int** apontado por **exp.** A função **frexp** retorna o valor **x,** tal que **x** é um **double** com magnitude no intervalo [1/2, 1] ou zero e **valor** é igual a **x** vezes 2 elevado à potência ***exp.** Se **valor** for zero, ambas as partes do resultado são zero.

double ldexp(double x, int exp);

Multiplica um número de ponto flutuante por uma potência inteira de 2. Pode ocorrer um erro de dimensão. A função **ldexp** retorna o valor de **x** vezes 2 elevado à potência **exp.**

double log(double x);

Calcula o logaritmo natural de **x.** Ocorre um erro de domínio se o argumento for negativo. Pode ocorrer um erro de dimensão se o argumento for zero.

double logl0(double x);

Calcula o logaritmo base 10 de **x.** Ocorre um erro de domínio se o argumento for negativo. Pode ocorrer um erro de dimensão se o argumento for zero.

double modf(double valor, double *iptr);

Divide o argumento **valor** em partes inteira e fracionária, tendo cada uma delas o mesmo sinal que o argumento. Ela armazena a parte inteira como **double** no objeto apontado por **iptr.** A função **modf** retorna a parte fracionária com sinal de **valor.**

double pow(double x, double y);

Calcula **x** elevado à potência **y.** Ocorre um erro de domínio se **x** for negativo e **y** não for um valor inteiro. Ocorre um erro de domínio se o resultado não puder ser representado quando **x** for zero e **y** for menor ou igual a zero. Pode ocorrer erro de dimensão.

double sqrt(double x);

Calcula a raiz quadrada não-negativa de **x.** Ocorre um erro de domínio se o argumento for negativo.

double ceil(double x);

Calcula o menor valor inteiro não menor que **x.**

double fabs(double x);

Calcula o valor absoluto de um número de ponto flutuante **x.**

double floor(double x);

Calcula o maior valor inteiro não maior que **x.**

double fmod(double x, double y);

Calcula o resto em ponto flutuante de **x/y.**

A.7 Desvios Externos <setjmp.h>

jmp_buf

Um tipo de array adequado para conter informações necessárias para restaurar um ambiente de chamada.

int setjmp(jmp_buf env);

Salva seu ambiente de chamada no argumento **jmp_buf** para uso posterior na função **longjmp.**

Se o retorno for de uma chamada direta, a macro **setjmp** retorna o valor zero. Se o retorno for de uma chamada para a função **longjmp,** a macro **setjmp** retorna um valor diferente de zero.

Uma chamada à macro **setjmp** só deve aparecer em um dos seguintes contextos:

• a expressão completa de controle de uma instrução de seleção ou iteração;

• um operando de um operador relacional ou de igualdade com o outro operando sendo uma expressão constante inteira, com a expressão resultante sendo a expressão completa de controle de uma instrução de seleção ou de iteração;

• o operando de um operador unário ! com a expressão resultante sendo a expressão completa de controle de uma instrução de seleção ou iteração;

• a expressão inteira de uma instrução de expressão.

void longjmp(jmp_buf env, int vai);

Restaura o ambiente salvo pela chamada mais recente da macro **setjmp** na mesma chamada do programa, com o argumento jm**p_buf** correspondente. Se não houver tal chamada, ou se a função que contém a chamada da macro **setjmp** tiver sua execução terminada nesse intervalo de tempo, o comportamento fica indefinido. Todos os objetos acessíveis possuem valores a partir do momento em que **longjmp** foi chamada, exceto que ficam indeterminados os valores dos objetos de duração automática de armazenamento que sejam locais à função que contém a chamada da macro **setjmp** correspondente e que não sejam do tipo volátil e foram modificados entre a chamada de **setjmp** e a chamada de **longjmp.**

Como ela evita a chamada normal da função e os mecanismos normais de retorno, **longjmp** deve ser executada corretamente no contexto de interrupções, sinais e quaisquer de suas funções associadas. Entretanto, se a função **longjmp** for chamada a partir de um manipulador de sinais aninhado (isto é, de uma função chamada como resultado de um sinal criado durante a manipulação de outro sinal), o comportamento fica indefinido.

Depois de **longjmp** ser concluído, a execução do programa continua como se a chamada correspondente da macro **setjmp** tivesse acabado de retornar o valor especificado por **val.** A função **longjmp** não pode fazer com que a macro **set jmp** retorne o valor 0; se **val** for 0, a macro **setjmp** retorna o valor 1.

A.8 Manipulação de Sinais <signal.h>

sig_atomic_t

O tipo inteiro de um objeto que pode ser acessado a partir de uma entidade atômica, mesmo na presença de interrupções assíncronas.

SIG_DFL SIG_ERR SIG_IGN

Essas geram expressões constantes com valores distintos que possuem tipos compatíveis com o tipo do segundo argumento e o valor de retorno da função **signal,** e cujo valor não se iguale ao endereço de qualquer função declarável; e a seguir, cada uma delas se expande em uma expressão constante inteira positiva que seja o número do sinal da condição especificada:

SIGABRT término anormal, tal como é iniciado pela função **abort.**
SIGFPE uma operação aritmética errada, tal como divisão por zero ou uma operação que resulte em overflow.
SIGILL detecção de uma imagem de função inválida, tal como uma instrução ilegal.
SIGINT recebimento de um sinal interativo de atenção.
SIGSEGV um acesso inválido ao armazenamento. **SIGTERM** uma solicitação de término enviada ao programa.

Uma implementação não precisa gerar qualquer um desses sinais, exceto como resultado de chamadas implícitas à função **raise.**

void (*signal(int sig, void (*func)(int)))(int);

Escolhe uma de três maneiras nas quais o recebimento do número do sinal **sig** deve ser manipulado. Se o valor de **func** for **SIG_DEF,** ocorrerá a manipulação default daquele sinal. Se o valor de **func** for **SIG_IGN,** o sinal será ignorado. Caso contrário, **func** apontará para uma função a ser chamada quando aquele sinal ocorrer.

Tal função é chamada um *manipulador de sinal.*

Quando ocorrer um sinal, se **func** apontar para uma função, em primeiro lugar o equivalente a **signal (sig, SIG_DFL)**; é executado ou é realizado um bloco do sinal, definido pela implementação. (Se o valor de **sig** for **SIGILL,** a ocorrência da redefinição de **SIG_DFL** é definida pela implementação.) A seguir, o equivalente a **(*func) (sig)**; é executado. A função **func** pode terminar por meio da execução da instrução **return** ou por meio da chamada das funções **abort, exit** ou **longjmp.** Se **func** executar uma instrução **return** e o valor de **sig** era **SIGFPE** ou qualquer outro valor definido pela implementação que correspondesse a uma exceção computacional, o comportamento fica indefinido. Caso contrário, o programa reiniciará a execução no ponto em que foi interrompido.

Se o sinal ocorrer de outra forma, que não como resultado da chamada da função **abort** ou **raise,** o comportamento fica indefinido se o manipulador de sinal chamar outra função na biblioteca padrão que não seja a própria função **signal** (com o primeiro argumento do número do sinal correspondendo ao sinal que causou a chamada do manipulador) ou se referir a qualquer objeto com duração de armazenamento estático que não seja atribuindo um valor a uma variável do tipo **volatile sig_atomic_t** para

duração de armazenamento estático. Além disso, se tal chamada à função **signal** resultar em um retorno **SIG_ERR,** o valor de **errno** fica indeterminado.

Na inicialização do programa, o equivalente a

signal(sig, SIG_IGN);

pode ser executado para alguns sinais selecionados de uma maneira definida pela implementação; o equivalente a

signalsig, SIG_DFL) ;

é executado para todos os outros sinais definidos pela implementação.

Se a solicitação pode ser atendida, a função **signal** retorna o valor de **func** para a chamada mais recente a **signal** para o sinal **sig** especificado. Caso contrário, um valor de **SIG_ERR** é retornado e um valor positivo é armazenado em **errno.**

int raise(int sig);

A função **raise** envia o sinal **sig** para o programa que está sendo executado. A função **raise** retorna um valor zero se for bem-sucedida e um valor diferente de zero se for malsucedida.

A.9 Argumentos de Variáveis <stdarg.h>

va_list

Um tipo adequado para conter informações necessárias pelas macros **va_start, va_arg** e **va_end.** Se for desejado acesso a esses argumentos variáveis, a função chamada declarará um objeto (chamado **ap** nessa seção) tendo tipo **va_list.** O objeto **ap** pode ser passado como um argumento para outra função; se aquela função chamar a macro **va_arg** com o parâmetro **ap,** o valor de **ap** na função de chamada é determinado e será passado para a macro **va_end** antes de qualquer referência posterior a **ap.**

void va_start (va_list ap, ***parmN);***

Será chamada antes de qualquer acesso a argumentos sem nome. A macro **va_start** inicializa **ap** para uso subseqüente por **va_arg** e **va__end.** O parâmetro *pannN é* o identificador do parâmetro da extremidade direita na lista de parâmetros de variáveis da definição da função (aquele imediatamente antes de , ...). Se o parâmetro *parmN* for declarado com a classe de armazenamento **register,** com uma função ou tipo de array, ou com um tipo que não seja compatível com o tipo que resulta após a aplicação das promoções default dos argumentos, o comportamento será indefinido.

tipo **va_arg(va_list ap,** *tipo);*

Gera uma expressão que tem o tipo e valor do próximo argumento na :hamada. O parâmetro **ap** será o mesmo que **va_list ap** inicializado por **va_start.** Cada chamada a **va_arg** modifica **ap,** de forma que os valores dos sucessivos argumentos são retornados seqüencialmente. O parâmetro *tipo é* o nome de um tipo especificado, de forma que o tipo de um ponteiro a um objeto que tenha o tipo especificado possa ser obtido simplesmente antecedendo *tipo* com *. Se não houver argumento seguinte, ou se *tipo* não for compatível com o tipo do próximo argumento (tal como se ele não foi promovido de acordo com as promoções default de argumentos), o comportamento da macro fica indefinido. A primeira chamada à macro **va_arg** depois da chamada à macro **va_start** retoma o valor do argumento após o especificado por *parmN.* Chamadas sucessivas retomam valores dos argumentos restantes em seqüência.

void va_end(va_list ap);

Facilita um retorno normal da função cuja lista de argumentos variáveis foi chamada pela expansão de **va_start** que inicializou **va_list ap.** A macro **va_end** pode modificar **ap** de forma que ele não seja mais utilizável (sem uma chamada intermediária a **va_start).** Se não houver chamada correspondente à macro **va_start,** ou se a macro **va_end** não for chamada antes do retorno, o comportamento da macro fica indefinido.

A. 10 Entrada/Saída <stdio.h>

_IOFBF
_IOLBF
_IONBF

Expressões constantes inteiras com valores distintos, adequadas para uso como terceiro argumento para a função **setvbuf.**

BUFSIZ
Uma expressão constante inteira, que tem o tamanho do buffer usado pela função **setbuf.**

EOF
Uma expressão constante inteira negativa que é retornada por várias funções para indicar o fim do arquivo (end-of-file), isto é, término da entrada de dados de um fluxo.

FILE
Um tipo de objeto capaz de gravar todas as informações necessárias para controlar um fluxo de dados, incluindo seu indicador de posição de arquivo, um ponteiro para seu buffer associado (se houver algum), um indicador de erro que registra se ocorreu um erro de leitura/gravação e um indicador de fim de arquivo que registra se o fim do arquivo foi alcançado.

FILENAME_MAX
Uma expressão constante inteira, que indica o tamanho necessário de um array de **char** para que ele seja grande o suficiente para conter a maior string de nome de arquivo que a implementação assegura que pode ser aberto.

FOPEN_MAX
Uma expressão constante inteira que indica o número mínimo de arquivos que a implementação assegura que podem ser abertos simultaneamente.

fpos_t
Um tipo de objeto capaz de gravar todas as informações necessárias para especificar de uma forma exclusiva todas as posições no interior de um arquivo.

L_tmpnam
Uma expressão constante inteira que indica o tamanho necessário de um array **char** para que ele seja grande o suficiente para conter uma string de nome de um arquivo temporário gerada pela função **tmpnam.**

NULL
Uma constante nula de ponteiro definida pela implementação.

SEEK_CUR SEEK_END SEEKSET
Expressões constantes inteiras com valores distintos, adequadas para uso como terceiro argumento da função **fseek.**

size_t

O tipo inteiro sem sinal do resultado do operador **sizeof.**

stderr

Expressão do tipo "ponteiro para **FILE"** que aponta para o objeto **FILE** associado ao fluxo de erro padrão.

stdin

Expressão do tipo "ponteiro para **FILE"** que aponta para o objeto **FILE** associado ao fluxo de entrada padrão.

stdout

Expressão do tipo "ponteiro para **FILE"** que aponta para o objeto **FILE** associado ao fluxo de saída padrão.

TMP_MAX

Expressão constante inteira que tem o número mínimo de nomes exclusivos de arquivos que devem ser gerados pela função **tmpnam.** O valor da macro **TMP_MAX** será no mínimo 25.

int remove(const char *filename);

Faz com que o arquivo cujo nome é a string apontada por **f ilename** não seja mais acessível por aquele nome. Uma tentativa subseqüente de abrir aquele arquivo usando aquele nome será malsucedida, a menos que ele seja criado novamente. Se o arquivo estiver aberto, o comportamento da função **remove** definido pela implementação. A função **remove** retorna zero se a operação for bem-sucedida, e um valor diferente de zero se a operação falhar.

int rename(const char *old, const char *new);

Faz com que o arquivo cujo nome é a string apontada por **old** seja, a partir desse momento, conhecido pelo nome apontado por **new.** O arquivo denominado **old** não fica mais acessível por aquele nome. Se um arquivo indicado pela string apontada por **new** já existir antes da chamada à função **rename,** o comportamento será definido pela implementação. A função **rename** retoma zero se a operação for bem-sucedida e um valor diferente de zero se ela falhar, caso em que, se o arquivo existia previamente, ainda será conhecido por seu nome original.

FILE *tmpfile(void);

Cria um arquivo binário temporário que será removido automaticamente quando for fechado ou no término do programa. Se o programa for encerrado de forma anormal, o fato de o arquivo temporário aberto ser removido ou não dependerá da implementação. O arquivo é aberto para atualização com o modo **"wb+**". A função **tmpfile** retorna um ponteiro para o fluxo do arquivo que é criado. Se o arquivo não puder ser criado, a função **tmpfile** retorna um ponteiro nulo.

char *tmpnam(char *s);

A função **tmpnam** gera uma string que é um nome válido de arquivo e que não seja nome de um arquivo existente. A função **tmpnam** gera uma string diferente cada vez que é chamada, até **TMP_MAX** vezes. Se ela for chamada mais que **TMP_MAX** vezes, o comportamento da função é definido pela implementação.

Se o argumento for um ponteiro nulo, a função **tmpnam** deixa seu resultado em um objeto estático interno e retorna um ponteiro para aquele objeto. As chamadas subseqüentes à função **tmpnam** podem modificar o mesmo objeto. Se o argumento não for um ponteiro nulo, assume-se que ele aponta para um array com pelo menos **L_tmpnam chars;** a função **tmpnam** escreve seu resultado naquele array e retorna o argumento como seu valor.

int fclose(FILE *stream);

A função **f close** faz com que o fluxo apontado por **stream** seja descarregado e o arquivo associado seja fechado. Quaisquer dados do fluxo, existentes no buffer e que não estejam gravados, são enviados para o ambiente do host para serem gravados no arquivo; quaisquer dados no buffer que não tenham sido lidos serão abandonados. O fluxo é desassociado do arquivo. Se o buffer associado foi alocado automaticamente, a alocação será desfeita. A função **f close** retorna zero se o fluxo for fechado satisfatoriamente ou **EOF** se for detectado algum erro.

int fflush(FILE *stream);

Se **stream** apontar para um fluxo de saída ou para um fluxo de atualização no qual a operação mais recente não foi entrada, a função **f f lush** faz com que quaisquer dados não-gravados daquele fluxo sejam enviados para o ambiente do host ou sejam gravados no arquivo; caso contrário, o comportamento fica indefinido.

Se **stream** for um ponteiro nulo, a função **f f lush** realiza a ação de descarga em todos os fluxos para os quais o comportamento foi definido anteriormente. A função **f f lush** retorna **EOF** se ocorrer um erro de gravação, e zero em caso contrário.

FILE *fopen(const char *filename, const char *mode);

A função **f open** abre um arquivo cujo nome é a string apontada por **f ilename** e associa um fluxo a ele. O argumento **mode** aponta para uma string que começa com uma das seqüências a seguir:
r abre arquivo de texto para leitura.
w trunca em comprimento zero ou cria arquivo de texto para gravação,
a anexa; abre ou cria arquivo de texto para gravação no final do arquivo,
rb abre arquivo binário para leitura.
wb trunca em comprimento zero ou cria arquivo binário para gravação.
ab anexa; abre ou cria arquivo binário para gravação no final do arquivo.
r+ abre arquivo-texto para atualização (leitura e gravação).
w+ trunca em comprimento zero ou cria arquivo-texto para atualização.
a+ anexa; abre ou cria arquivo-texto para atualização, gravando no final do arquivo.
r+b ou **rb+** abre arquivo binário para atualização (leitura^e gravação).
w+b ou **wb+** trunca em comprimento zero ou cria arquivo binário para atualização.
a+b ou **ab+** anexa; abre ou cria arquivo binário para atualização, gravando no final do arquivo.

Abrir um arquivo no modo de leitura (' **r** ' como primeiro caractere do argumento **mode)** falha se o arquivo não existir ou não puder ser lido. Abrir um arquivo com o modo de anexação (' **a** ' como o primeiro caractere do argumento **mode)** faz com que todas as gravações subseqüentes do arquivo sejam colocadas no final do arquivo atual, independentemente das chamadas intermediárias à função **f seek.** Em algumas implementações, abrir um arquivo binário com o modo de anexação (' **b**' como segundo ou terceiro caractere na lista anterior de valores do argumento **mode)** pode colocar

inicialmente o indicador de posição para o fluxo além do último dado gravado, tendo em vista a existência do caractere nulo.

Quando um arquivo é aberto no modo de atualização (' + [1] como segundo ou terceiro caractere na lista anterior de valores do argumento **mode),** tanto a entrada quanto a saída de dados podem ser realizadas no fluxo associado. Entretanto, a saída não pode ser seguida diretamente pela entrada sem uma chamada inter-
mediaria à função **fflush** ou a uma função de posicionamento do arquivo **(f seek, f setpos** ou **rewind),** e a entrada não pode ser seguida diretamente da saída sem uma chamada intermediária a uma função de posicionamento de arquivo, a menos que a operação de entrada encontre o fim do arquivo. Abrir (ou criar) um arquivo de texto com o modo de atualização pode, em vez disso, abrir (ou criar) um fluxo binário em algumas implementações.

Ao ser aberto, o fluxo é colocado completamente no buffer se e somente se puder ser determinado que ele não se refira a um dispositivo interativo. Os indicadores de erro e fim de arquivo (end-of-file) para o fluxo são zerados. A função **fopen** retorna um ponteiro para o objeto que controla o fluxo. Se a operação de abertura falhar, **f open** retorna um ponteiro nulo.

FILE *freopen(const char *filename, const char *mode, FILE *stream);

A função **freopen** abre o arquivo cujo nome é a string apontada por **filename** e associa a ele o fluxo apontado por **stream.** O argumento **mode** é usado exatamente da mesma forma que a função **fopen.**

A função **freopen** tenta inicialmente fechar qualquer arquivo que esteja associado ao fluxo especificado. Uma falha em fechar o arquivo é ignorada. Os indicadores de erro e do fim do arquivo para o fluxo são zerados. A função **f reopen** retorna um ponteiro nulo se a operação de abertura falhar. Caso contrário, **f reopen** retorna o valor de **stream.**

void setbuf(FILE *stream, char *buf);

A função **setbuf** é equivalente à função **setvbuf** chamada com os valores **_IOFBF** para **mode** e **BUFSIZ** para **size** ou (se **buf** for um ponteiro nulo) com o valor **_IONBF** para **mode.** A função **setbuf** não retorna valor algum.

int setvbuf(FILE *stream, char *buf, int mode, size_t size);

A função **setvbuf** só pode ser usada depois de o fluxo apontado por **stream** ter sido associado a um arquivo aberto e antes de qualquer outra operação ser realizada no fluxo. O argumento **mode** determina como **stream** será colocado no buffer, da forma que se segue: **_I0FBF** faz com que entrada/saída sejam colocadas completamente no buffer; **_IOLBF** faz com que a entrada/saída sejam colocadas no buffer em linhas; **_I0NBF** faz com que a entrada/saída não sejam colocadas no buffer. Se **buf** não for um ponteiro nulo, o array para o qual ele aponta pode ser usado no lugar de um buffer alocado pela função **setvbuf.** O argumento **size** especifica o tamanho do array. Em qualquer instante, o conteúdo do array é indeterminado. A função **setvbuf** retorna zero ao ser bem-sucedida ou um valor diferente de zero se houver um valor inválido para **mode** ou se a solicitação não puder ser atendida.

int fprintf(FILE *stream, const char *format, ...);

A função **f printf** grava a saída no fluxo apontado por **stream,** sob controle da string apontada por **format,** que especifica como os argumentos subseqüentes são

convertidos para a saída. Se houver argumentos insuficientes para o formato, o comportamento da função fica indefinido. Se o formato for esgotado e ainda houver argumentos remanescentes, os argumentos em excesso são calculados (como sempre) mas são ignorados. A função **fprintf** retorna quando o final da string de formato é encontrado. Veja o Capítulo 9, "Formatação de Entrada/Saída" para obter uma descrição detalhada das especificações de conversão. A função **fprintf** retorna o número de caracteres transmitidos ou um valor negativo se ocorrer um erro de saída.

int fscanf(FILE *stream, const char *format, ...);

A função **f scanf** lê a entrada de um fluxo apontado por **stream,** sob o controle da string apontada por **format,**que especifica as seqüências de entrada admissíveis e como elas são convertidas para atribuição, usando os argumentos subseqüentes como ponteiros a objetos para receber a entrada formatada. Se não houver argumentos suficientes para o formato, o comportamento fica indefinido. Se o formato terminar e permanecerem argumentos, os argumentos em excesso são calculados (como sempre) mas são ignorados. Veja o Capítulo 9, "Formatação de Entrada/Saída" para obter uma descrição detalhada das especificações de entrada.

A função **f scanf** retorna o valor da macro **EOF** se ocorrer uma falha de entrada antes de qualquer conversão. Caso contrário, a função **f scanf** retorna o número de itens de entrada atribuídos, que pode ser menor do que o de itens fornecidos, ou mesmo zero, no caso de uma falha prematura de correspondência.

int printf(const char *format, ...);

A função **printf** é equivalente a **f printf** com o argumento **stdout** interposto antes dos argumentos de **printf.** A função **printf** retorna o número de caracteres transmitidos, ou um valor negativo se ocorrer um erro de saída.

int scanf(const char *format, ...);

A função **scanf** é equivalente a **f scanf** com o argumento **stdin** interposto antes dos argumentos de **scanf.**A função **scanf** retorna o valor da macro **EOF** se ocorrer uma falha de entrada antes de qualquer conversão. Caso contrário, **scanf** retorna o número de itens de entrada atribuídos, que pode ser menor do que o de itens fornecidos, ou mesmo zero, no caso de uma falha prematura de correspondência.

int sprintf(char *s, const char *format, ...);

A função **sprintf** é equivalente a **f printf**, exceto que o argumento **s** especifica um array no qual a saída gerada deve ser gravada, em vez de ser enviada para um dispositivo (ou fluxo) de saída. Um caractere nulo é gravado no final dos caracteres gravados; ele não é contado como parte do total retornado. O comportamento de cópia entre objetos que se superpõem é indefinido. A função **sprintf** retorna o número de caracteres gravados no array, não contando o caractere nulo de terminação.

int sscanf(const char *s, const char *format, ...);

A função **sscanf** é equivalente a **f scanf**, exceto que o argumento **s** especifica uma string da qual a entrada é obtida, em vez de ser obtida de um dispositivo (ou fluxo) de entrada. Alcançar o fim da string é equivalente a encontrar o final do arquivo na função **f scanf**. Se acontecer a cópia entre objetos que se superpõem, o comportamento da função fica indefinido.

A função **sscanf** retorna o valor da macro **EOF** se ocorrer uma falha de entrada antes de qualquer conversão. Caso contrário, a função **sscanf** retorna o número de itens

de entrada atribuídos, que pode ser menor do que o de itens fornecidos, ou mesmo zero, no caso de uma falha prematura de correspondência.

int vfprintf(FILE *stream, const char *format, va_list arg);

A função **vf printf** é equivalente a **f printf**, com a lista de variáveis dos argumentos substituída por **arg,** que deve ter sido inicializado pela macro **va_start** (e possivelmente pelas chamadas subseqüentes a **va_arg).**

A função **vf printf** não chama a macro **va_end.** A função **vf printf** retorna o número de caracteres transmitidos, ou um valor negativo se ocorrer erro de saída.

int vprintf (const char *format, va_list arg);

A função **vprintf** é equivalente a **printf,** com a lista de variáveis dos argumentos substituída por **arg,** que deve ter sido inicializada pela macro **va_start** (e possivelmente por chamadas subseqüentes a **va_arg).** A função **vprintf** não chama a macro **va_end.** A função **vprintf** retorna o número de caracteres transmitidos, ou um valor negativo se ocorrer erro de saída.

int vsprintf(char *s, const char *format, va_list arg);

A função **vsprintf** é equivalente a **sprintf,** com a lista de variáveis dos argumentos substituída por **arg,** que deve ter sido inicializado pela macro **va_start** (e possivelmente por chamadas subseqüentes a **va_arg).** A função **vsprintf** não chama a macro **va_end.** Se acontecer a cópia entre objetos que se superpõem, o comportamento da função fica indefinido. A função **vsprintf** retorna o número de caracteres gravados no array, sem contar o caractere nulo de terminação.

int fgetc(FILE *stream);

A função **f getc** obtém o próximo caractere (se houver) como um **unsigned char** convertido a um **int,** do fluxo de entrada apontado por **stream,** e avança o indicador de posição do arquivo associado para o fluxo (se houver). A função **f getc** retorna o próximo caractere do fluxo de entrada apontado por **stream.** Se o fluxo estiver no fim do arquivo, é estabelecido o indicador de fim de arquivo para o fluxo e **f getc** retorna **EOF.** Se ocorrer um erro de leitura, é estabelecido o indicador de erros para o fluxo e **fgetc** retorna **EOF.**

char *fgets(char *s, int n, FILE *stream);

A função **fgets** lê no máximo um caractere a menos que o número de caracteres especificado por **n** no fluxo apontado por **stream** no array apontado por **s.** Nenhum caractere adicional é lido depois de um caractere de nova linha (que é retido) ou depois de um final de arquivo. Um caractere nulo é gravado imediatamente após o último caractere lido no array.

A função **fgets** retorna **s** se for bem-sucedida. Se o final do arquivo for encontrado e nenhum caractere tiver sido lido no array, o conteúdo do array permanece inalterado e um ponteiro nulo é retornado. Se ocorrer um erro de leitura durante a operação, o conteúdo do array fica indeterminado e é retornado um ponteiro nulo.

int fputc(int c, FILE *stream);

A função **fputc** grava o caractere especificado por **c** (convertido em um **unsigned char)** no fluxo de saída apontado por **stream,** no local estabelecido pelo indicador de posição do arquivo para o fluxo (se definido), e avança o indicador apropriadamente. Se o arquivo não puder atender às solicitações de posicionamento, ou se o fluxo foi aberto no modo de anexação, o caractere é anexado ao fluxo de saída. A

função **fputc** retorna o caractere gravado. Se ocorrer um erro de gravação, o indicador de erro para o fluxo é definido e **fputc** retorna **EOF.**

int fputs(const char *s, FILE *stream);

A função **f puts** grava a string apontada por **s** no fluxo apontado por **stream.** O caractere nulo de terminação não é gravado. A função **fputs** retorna **EOF** se ocorrer um erro de gravação; caso contrário retorna um valor não-negativo.

int getc(FILE *stream);

A função **getc** é equivalente a **fgetc,** exceto que se ela for implementada como uma macro pode avaliar **stream** mais de uma vez — o argumento deve ser uma expressão sem efeitos secundários.

A função **getc** retorna o próximo caractere do fluxo de entrada apontado por **stream.** Se o fluxo estiver no final do arquivo, é estabelecido o indicador de final de arquivo para o fluxo e **getc** retorna **EOF.** Se ocorrer um erro de leitura, é estabelecido o indicador de erro para o fluxo e **getc** retorna **EOF.**

int getchar(void) ;

A função **getchar** é equivalente a **getc** com o argumento **stdin.** A função **getchar** retorna o próximo caractere do fluxo de entrada apontado por **stdin.** Se o fluxo estiver no final do arquivo, é estabelecido o indicador de fim de arquivo para o fluxo e **getchar** retorna **EOF.** Se ocorrer um erro de leitura, é estabelecido o indicador de erro para o fluxo e **getchar** retorna **EOF.**

char *gets(char *s);

A função **gets** lê caracteres do fluxo de entrada apontado por **stdin,** no array apontado por **s,** até que o final do arquivo seja encontrado ou seja lido um caractere de nova linha. Qualquer caractere de nova linha é ignorado e um caractere nulo é gravado imediatamente depois do último caractere lido no array. A função **gets** retorna **s** se for bem-sucedida. Se o final do arquivo for encontrado e nenhum caractere for lido no array, o conteúdo do array permanece inalterado e um ponteiro nulo é retornado. Se ocorrer um erro de leitura durante a operação, o conteúdo do array fica indeterminado e é retornado um ponteiro nulo.

int putc(int c, FILE *stream);

A função **putc** é equivalente a **f putc,** exceto que, se for implementada como uma macro, pode avaliar **stream** mais de uma vez, portanto os argumentos nunca devem ser uma expressão com efeitos secundários. A função **putc** retorna o caractere gravado. Se ocorrer um erro de gravação, é estabelecido o indicador de erro do fluxo e **putc** retorna **EOF.**

int putchar(int c);

A função **putchar** é equivalente a **putc** como segundo argumento **stdout.** A função **putchar** retorna o caractere gravado. Se ocorrer um erro de gravação, é estabelecido o indicador de erro para o fluxo e **putchar** retorna **EOF.**

int puts(const char *s) ;

A função **puts** grava a string apontada por **s** no fluxo apontado por **stdout** e adiciona um *caractere de nova* linha à saída. O caractere nulo de terminação não é gravado. A função **puts** retorna **EOF** se ocorrer um erro de gravação; caso contrário retorna um valor não-negativo.

int ungetc(int c, FILE *stream);

A função **ungetc** coloca o caractere especificado por **c** (convertido em um **unsigned char)** de volta no fluxo de entrada apontado por **stream.** Os caracteres colocados de volta serão retornados pelas leituras subseqüentes naquele fluxo, na ordem inversa de sua colocação. Uma chamada intermediária (com o fluxo apontado por **stream)** bem-sucedida a uma função de posicionamento do arquivo **(fseek, fsetpos** ou **rewind)** descarta quaisquer caracteres colocados de volta no fluxo. O armazenamento externo correspondente ao fluxo fica inalterado.

Garante-se a volta de um caractere. Se a função **ungetc** for chamada muitas vezes no mesmo fluxo sem uma leitura intermediária ou operação intermediária de posicionamento de arquivo naquele fluxo, a operação pode falhar. Se o valor de **c** for igual ao da macro **EOF,** a operação falha e o fluxo de entrada permanece inalterado.

Uma chamada bem-sucedida à função **ungetc** limpa o indicador de final de arquivo do fluxo. O valor do indicador de final de arquivo para o fluxo, depois da leitura ou do descarte de todos os caracteres colocados de volta, deve ser o mesmo de antes da volta dos caracteres. Para um fluxo de texto, o valor de seu indicador de posição de arquivo depois de uma chamada bem-sucedida à função **ungetc** não é especificado até que todos os caracteres colocados de volta sejam lidos ou descartados. Para um fluxo binário, seu indicador de posição de arquivo é determinado por cada chamada bem-sucedida à função **ungetc;** se seu valor era zero antes de uma chamada, fica indeterminado após a chamada. A função **ungetc** retorna o caractere colocado de volta, depois da conversão, ou **EOF** se a operação falhar.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

A função **f read** lê, no array apontado por **ptr,** até **nmemb** elementos cujo tamanho é especificado por **size,** do fluxo apontado por **stream.** O indicador de posição de arquivo para o fluxo (se definido) é avançado de acordo com o número de caracteres cuja leitura foi bem-sucedida. Se ocorrer um erro, o valor resultante do indicador de posição de arquivo para o fluxo fica indeterminado. Se um elemento parcial for lido, seu valor fica indeterminado.

A função **fread** retoma o número de elementos que foram lidos satisfatoriamente, que pode ser menor do que **nmeirib** se ocorrer um erro ou o final do arquivo for encontrado. Se **size** ou **nmemb** forem zero, **fread** retorna zero e o conteúdo do array e o estado do fluxo permanecem inalterados.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

A função **fwrite** grava no fluxo de entrada apontado por **stream,** a partir do array apontado por **ptr,** até **nmemb** elementos cujo tamanho é especificado por **size.** O indicador de posição de arquivo para o fluxo (se definido) é avançado de acordo com o número de caracteres cuja gravação foi bem-sucedida. Se ocorrer um erro, o valor resultante da posição do arquivo para o fluxo fica indeterminado. A função **f write** retorna o número de elementos gravados satisfatoriamente, que só pode ser menor do que **nmemb** se ocorrer um erro de gravação.

int fgetpos(FILE *stream, fpos_t *pos);

A função **f getpos** armazena, no objeto apontado por **pos,** o valor atual do indicador de posição do arquivo para o fluxo apontado por **stream.** O valor armazenado

contém informações não-especificadas que podem ser utilizadas pela função **f setpos** para reposicionar o fluxo em seu local na ocasião da chamada à função **fgetpos.** Se bem-sucedida, a função **fgetpos** retorna zero; em caso de falha, a função **f getpos** retorna um valor diferente de zero e armazena em **errno** um valor positivo, definido de acordo com a implementação.

int fseek(FILE *stream, long int offset, int whence);

A função **fseek** define o indicador de posição de arquivo para o fluxo apontado por **stream.** Para um fluxo binário, a nova posição, medida em caracteres a partir do início do arquivo, é obtida adicionando **offset** à posição especificada por **whence.** A posição especificada é o início do arquivo se **whence** for **SEEK_SET,** o valor atual do indicador de posição do arquivo se for **SEEK_CUR** e o final do arquivo se for **SEEK_END.** Um fluxo binário não precisa suportar significativamente as chamadas a **f seek** com um valor de **SEEKEND** para **whence.** Para um fluxo de texto, ou **offset** deve ser zero, ou **offset** deve ser um valor retornado por uma chamada anterior à função **ftell** no mesmo fluxo e **whence** deve ser **SEEK_SET.**

Uma chamada bem-sucedida à função **fseek** limpa o indicador de final de arquivo para o fluxo e desfaz quaisquer efeitos da função **ungetc** no mesmo fluxo. Depois de uma chamada a **f seek,** a próxima operação em um fluxo de atualização pode ser tanto entrada quanto saída. A função **f seek** só retorna um valor diferente de zero se uma solicitação não puder ser satisfeita.

int fsetpos(FILE *stream, const fpos_t *pos);

A função **fsetpos** define o indicador de posição do arquivo para o fluxo apontado por **stream,** de acordo com o valor do objeto apontado por **pos,** que deve ser um valor obtido de uma chamada anterior à função **fgetpos** no mesmo fluxo. Uma chamada bem-sucedida à função **fsetpos** limpa o indicador de final de arquivo para o fluxo e desfaz quaisquer efeitos da função **ungetc** no mesmo fluxo. Depois de uma chamada a **fsetpos,** a próxima operação em um fluxo de atualização pode ser tanto uma entrada quanto uma saída. Se bem-sucedida, a função **fsetpos** retorna zero; em caso de falha, a função **f setpos** retorna um valor diferente de zero e armazena em **errno** um valor positivo, definido de acordo com a implementação.

long int ftell(FILE *stream);

A função **ftell** obtém o valor atual do indicador da posição do arquivo para o fluxo apontado por **stream.**

Para um fluxo binário, o valor é o número de caracteres desde o início do arquivo. Para um fluxo de texto, seu indicador de posição do arquivo contém informações não-especificadas, utilizáveis pela função **fseek** para retornar o indicador de posição do arquivo a sua posição por ocasião da chamada a **ftell;** a diferença entre dois de tais valores de retorno não é necessariamente uma medida significativa do número de caracteres lidos ou gravados. Se bem-sucedida, a função **ftell** retorna o valor atual do indicador de posição do arquivo para o fluxo. Em caso de falha, a função **ftell** retorna **-1L** e armazena um em **errno** valor positivo definido de acordo com a implementação.

void rewind(FILE *stream);

A função **rewind** define no início do arquivo seu indicador de posição para o fluxo apontado por **stream.** Ela é equivalente a **(void) fseek(stream, OL, SEEK_SET)** exceto que o indicador de erros para o fluxo também é reinicializado.

void clearerr(FILE *stream);

A função **clearerr** limpa os indicadores de fim de arquivo e de erros para o fluxo apontado por **stream.**

int feof(FILE *stream);

A função **f eof** faz um teste com o indicador de fim de arquivo para o fluxo apontado por **stream.** A função **feof** retorna um valor diferente de zero se e somente se for definido o indicador de final de arquivo para **stream.**

int ferror(FILE *stream);

A função **f error** testa o indicador de erro para o fluxo apontado por **stream.** A função **f error** retorna um valor diferente de zero se e somente se for definido o indicador de erros para **stream.**

void perror(const char *s);

A função **perror** mapeia o número do erro na expressão inteira **errno** para uma mensagem de erro. Ela escreve uma seqüência de caracteres no fluxo padrão de erro da seguinte forma: em primeiro lugar (se **s** não for um ponteiro nulo e o caractere apontado por **s** não for um caractere nulo), a string apontada por **s** seguida de dois pontos (:) e um espaço; depois uma string apropriada de mensagem de erros seguida de um caractere de nova linha. Os conteúdos das strings de mensagens de erro são os mesmos retornados pela função **strerror** com o argumento **errno,** que são definidoS-pela implementação.

A.11 Utilidades Gerais <stdlib.h>

EXIT_FAILURE
EXIT_SUCCESS

Expressões inteiras que podem ser usadas como argumento da função **exit** para retornar o status de terminação mal ou bem-sucedida, respectivamente, para o ambiente principal.

MB_CUR_MAX

Uma expressão inteira positiva cujo valor é o número máximo de bytes em um caractere de vários bytes para o conjunto de caracteres especificado pelo local atual (categoria **LC_CTYPE)** e cujo valor nunca é maior do que **MB_LEN_MAX.**

NULL

Uma constante de ponteiro nula definida conforme a implementação.

RAND_MAX

Uma expressão de intervalo constante, cujo valor é o máximo retornado pela função **rand.** O valor da macro **RAND_MAX** deve ser no mínimo 32767.

div_t

Um tipo de estrutura que é o tipo do valor retornado pela função **div.**

ldiv_t

Um tipo de estrutura que é o tipo do valor retornado pela função **ldiv.**

size_t

O tipo inteiro sem sinal do resultado do operador **sizeof.**

wchar_t

Um tipo inteiro cujo intervalo de valores pode representar códigos distintos para todos os membros do maior conjunto estendido de caracteres especificado entre os locais suportados; o caractere nulo deve ter valor de código zero e cada membro do conjunto básico de caracteres deve ter um valor de código igual ao seu valor quando usado como caractere isolado em uma constante inteira de caracteres.

double atof(const char *nptr);

Converte para representação **double** a parte inicial da string apontada por **nptr.** A função **atof** retorna o valor convertido.

int atoi(const char *nptr);

Converte para representação **int** a parte inicial da string apontada por **nptr.** A função **atoi** retorna o valor convertido.

long int atoi(const char *nptr);

Converte para representação **long** a parte inicial da string apontada por **nptr.** A função **atoi** retorna o valor convertido.

double strtod(const char *nptr, char **endptr);

Converte para representação **double** a parte inicial da string apontada por **nptr.** Em primeiro lugar, ela decompõe a string de entrada em três partes: uma seqüência inicial, possivelmente vazia, de caracteres de espaço em branco (conforme o especificado pela função **isspace),** uma seqüência central, semelhante a uma constante de ponto flutuante; e uma string final de um ou mais caracteres não-reconhecidos, incluindo o caractere nulo de terminação da string de entrada. A seguir, ela tenta converter a seqüência central em um número de ponto flutuante e retorna o resultado. A forma expandida da seqüência central consiste em um sinal opcional de adição ou subtração, seguido de uma seqüência não vazia de dígitos contendo opcionalmente um caractere de ponto decimal e de uma parte exponencial opcional, sem sufixo flutuante. A seqüência central é definida como a subseqüência inicial mais longa da string de entrada que comece com o primeiro caractere diferente de um caractere de espaço em branco e que seja da forma esperada. A seqüência central não possuirá caracteres se a string de entrada estiver vazia ou consistir inteiramente em caracteres de espaço em branco ou se o primeiro caractere diferente de um caractere de espaço em branco for diferente de um sinal, um dígito ou um ponto decimal.

Se a seqüência central tiver a forma esperada, a seqüência de caracteres começando com o primeiro dígito ou caractere de ponto decimal (o que ocorrer em primeiro lugar) é interpretada como uma constante de ponto flutuante, a menos que o caractere de ponto decimal seja usado no lugar de um ponto final e que se não aparecer uma parte exponencial nem um caractere de ponto decimal, assume-se que um ponto decimal vem após o último caractere da string. Se a seqüência central começar com um sinal de subtração, o valor resultante da conversão é negativo. Um ponteiro para a string final é armazenado no objeto apontado por **endptr,** contanto que **endptr** não seja um ponteiro nulo.

Se a seqüência central estiver vazia ou não tiver a forma esperada, nenhuma conversão é realizada; o valor de **nptr** é armazenado no objeto apontado por **endptr,** contanto que **endptr** não seja um ponteiro nulo.

A função **strtod** retorna o valor convertido, se houver. Se nenhuma conversão puder ser realizada, é retornado o valor zero. Se o valor correto estiver além do limite dos valores que podem ser representados, o valor positivo ou negativo de **HUGE_VAL** é retornado (de acordo com o sinal do valor correto) e o valor da macro **ERANGE** é armazenado em **errno.** Se o valor correto for causar underflow, é retornado o valor zero e o valor da macro **ERANGE** é armazenado em **errno.**

long int strtol(const char *nptr, char **endptr, int base);

Converte para representação **long int** a parte inicial da string apontada por **nptr.** Em primeiro lugar, ela decompõe a string de entrada em três partes: uma seqüência inicial, possivelmente vazia, de caracteres de espaço em branco (conforme o especificado pela função **isspace),** uma seqüência central, semelhante a um inteiro representado em alguma base determinada pelo valor de **base;** e uma string final de um ou mais caracteres não-reconhecidos, incluindo o caractere nulo de terminação da string de entrada. A seguir, ela tenta converter a seqüência central em um inteiro e retorna o resultado.

Se o valor de **base** for zero, a forma esperada da seqüência central é a de uma constante inteira, precedida opcionalmente por um sinal de adição ou subtração, mas

não incluindo um sufixo inteiro. Se o valor de **base** estiver entre 2 e 36, a forma esperada da seqüência central é uma seqüência de letras e dígitos representando um inteiro com base especificada por **base,** precedida opcionalmente por um sinal de adição ou subtração, mas não incluindo um sufixo inteiro. Os valores 10 a 35 são atribuídos às letras de **a** (ou **A)** a **z** (ou **Z);** apenas as letras cujos valores atribuídos forem menores do que o de **base** são permitidas. Se o valor de **base** for 16, os caracteres **Ox** ou **0X** podem preceder opcionalmente a seqüência de letras e dígitos, após o sinal de adição ou subtração, se ele estiver presente.

A seqüência central é definida como a subseqüência inicial mais longa da string de entrada que comece com o primeiro caractere diferente de um caractere de espaço em branco e que seja da forma esperada. A seqüência central não possuirá caracteres se a string de entrada estiver vazia ou consistir inteiramente em caracteres de espaço em branco ou se o primeiro caractere diferente de um espaço em branco for diferente de um sinal ou de uma letra ou dígito permitido.

Se a seqüência central tiver a forma esperada e o valor de **base** for zero, a seqüência de caracteres começando com o primeiro dígito é interpretada como uma constante inteira. Se a seqüência central tiver a forma esperada e o valor de **base** estiver entre 2 e 36, ele é usado como base para conversão, atribuindo a cada letra seu valor da forma descrita anteriormente. Se a seqüência central começar com um sinal de subtração, o valor resultante da conversão é negativo. Um ponteiro para a string final é armazenado no objeto apontado por **endptr,** contanto que **endptr** não seja um ponteiro nulo.

Se a seqüência central estiver vazia ou não tiver a forma esperada, nenhuma conversão é realizada; o valor de **nptr** é armazenado no objeto apontado por **endptr,** contanto que **endptr** não seja um ponteiro nulo.

A função **strtol** retorna o valor convertido, se houver. Se nenhuma conversão puder ser realizada, é retornado o valor zero. Se o valor correto estiver além do limite dos valores que podem ser representados, **LONG_MAX** ou **LONG_MIN** é retornado (de acordo com o sinal do valor correto) e o valor da macro **ERANGE** é armazenado em **errno.**

unsigned long int strtoul(const char *nptr, char **endptr, int base);

Converte para representação **unsigned long int** a parte inicial da string apontada por **nptr.** A função **strtoul** funciona de forma idêntica à função **strtol.** A função **strtoul** retorna o valor convertido, se houver. Se nenhuma conversão puder ser realizada, é retornado o valor zero. Se o valor correto estiver além do limite dos valores que podem ser representados, **ULONG_MAX** é retornado e o valor da macro **ERANGE** é armazenado em **errno.**

int rand(void);

A função **rand** calcula uma seqüência de inteiros pseudo-aleatórios no intervalo de 0 a **RANDMAX.** A função **rand** retorna um inteiro pseudo-aleatório.

void srand(unsigned int seed);

Usa o argumento como semente (ou seed) de uma nova seqüência de números pseudo-aleatórios a ser retornado pelas chamadas subseqüentes a **rand.** Assim sendo, se **rand** for chamada com o mesmo valor de semente, a seqüência de números pseudo-

aleatórios será repetida. Se **rand** for chamada antes de qualquer chamada a **srand** ter sido feita, será gerada a mesma seqüência de quando **srand** é chamada com valor de semente igual a 1. As funções a seguir definem uma implementação portátil de **rand** e **srand.**

```
static unsigned long int next = 1;
int rand (void)  /* assume-se RAND_MAX igual a 32767 */
{
next = next * 1103515245 + 12345;
return (unsigned int)   (next/65536) % 32768;
}
void srand(unsigned int seed) {
next = seed;
}
```

void *calloc(size_t nmernb, size_t size);

Aloca espaço para um array de **nmernb** objetos, tendo cada um deles o tamanho definido por **size.** O espaço é inicializado como zero para todos os bits. A função **calloc** retorna tanto um ponteiro nulo quanto um ponteiro para o espaço alocado.

void free(void *ptr);

Faz com que a alocação do espaço apontado por **ptr** seja removida, isto é, torna o espaço disponível para alocação posterior. Se **ptr** for um ponteiro nulo, não acontece ação alguma. Caso contrário, se o argumento não corresponder a um ponteiro retornado anteriormente pelas funções **calloc, malloc** e **realloc,** ou se o espaço foi desalocado por uma chamada a **f ree** ou **realloc,** o comportamento da função fica indefinido.

void *malloc(size_t size);

Aloca espaço para um objeto cujo tamanho é especificado por **size** e cujo valor é indeterminado. A função **malloc** retorna um ponteiro nulo ou um ponteiro para o espaço alocado.

void *realloc(void *ptr, size_t size);

Muda o tamanho do objeto apontado por **ptr** para o tamanho especificado por **size.** O conteúdo do objeto ficará inalterado até o menor entre os tamanhos novo e antigo. Se o novo tamanho for maior, o valor da parte do objeto alocada posteriormente fica indefinido. Se **ptr** for um ponteiro nulo, a função **realloc** age como a função **malloc** para o tamanho especificado. Caso contrário, se **ptr** não corresponder a um ponteiro retornado anteriormente pelas funções **calloc, malloc** ou **realloc,** ou se o espaço foi desalocado por uma chamada às funções **free** ou **realloc,** o comportamento da função fica indefinido. Se o espaço não puder ser desalocado, o objeto apontado por **ptr** fica inalterado. Se **size** for zero e **ptr** não for nulo, o objeto para o qual **ptr** aponta é liberado. A função **real loc** retorna tanto um ponteiro nulo quanto um ponteiro para o espaço alocado, possivelmente modificado.

void abort(void) ;

Faz com que ocorra um encerramento anormal do programa, a menos que o sinal **SIGABRT** esteja sendo captado e o manipulador de sinal não faça o retorno. O modo de implementação determinará se os fluxos abertos de saída são enviados, se os fluxos abertos são fechados e se os arquivos temporários são removidos. Uma forma de status de *encerramento insatisfatório {unsuccessful termination),* definida pela

implementação, é retornada para o ambiente principal por meio da chamada à função **raise (SIGABRT)**. A função **abort** não pode retornar a quem fez a chamada.

int atexit(void (*fune)(void));

Registra a função apontada por **f une** para ser chamada sem argumentos no encerramento normal do programa.

A implementação fornecerá suporte para o registro de, no mínimo, 32 funções. A função **atexit** retorna zero se o registro for bem-sucedido e um valor diferente de zero se falhar.

void exit(int status);

Faz com que ocorra um encerramento normal do programa. Se um programa executar mais de uma chamada a **exit,** o comportamento da função fica indefinido. Em primeiro lugar todas as funções registradas pela função **atexit** são chamadas na ordem inversa de seu registro. Cada função é chamada tantas vezes quanto foi registrada. A seguir, todos os fluxos abertos com dados não-gravados existentes no buffer são descarregados, todos os fluxos abertos são fechados e todos os arquivos criados pela função **tmpfile** são removidos.

Finalmente, o controle é retomado ao ambiente principal. Se o valor de **status** for zero ou **EXIT_SUCCESS,** uma forma de *encerramento satisfatório* (ou *bem-sucedido),* definida pela implementação, é retomada. Se o valor de **status** for **EXIT_FAILURE,** uma forma de *encerramento não-satisfatório* (ou *malsucedido),* definida pela implementação, é retornada. Caso contrário, o status retornado é definido pela implementação. A função **exit** não pode retomar a quem a chamou.

char *getenv(const char *name);

Procura, em uma *lista de ambientes (environment list)* fornecida pelo ambiente principal, uma string que se iguale à string apontada por **name.** O conjunto de nomes de ambientes e o método para alterar a lista de ambientes são definidos pela implementação. Retorna um ponteiro para uma string associada ao membro da lista para o qual igualdade foi verificada. A string apontada não será modificada pelo programa, mas pode ser sobrescrita por uma chamada subseqüente à função **getenv.** Se o nome **(name)** especificado não puder ser encontrado, um ponteiro nulo é retomado.

int system(const char *string);

Passa a string apontada por **string** para o ambiente principal, para ser executada por um *processador de comandos* de uma forma definida pela implementação. Um ponteiro nulo pode ser usado para **string** verificar se existe um processador de comandos. Se o argumento for um ponteiro nulo, a função **system** retorna um valor diferente de zero somente se houver um processador de comandos disponível. Se o argumento não for um ponteiro nulo, a função **system** retoma um valor que varia conforme a implementação.

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Procura, em um array de **nmemb,** objetos cujo elemento inicial é apontado por **base,** um elemento que se iguale ao objeto apontado pela chave **(key).** O tamanho de cada elemento do array é especificado por **size.** A função de comparação apontada por **compar** é chamada com dois argumentos que apontam para o objeto **key** e para um elemento do array, naquela ordem. A função retomará um inteiro menor que, igual a ou

maior que zero se o objeto **key** for considerado, respectivamente, menor que, igual a ou maior que o elemento do array. O array consistirá em: todos os elementos que sejam menores, todos os elementos que sejam iguais e todos os elementos que sejam maiores que o objeto **key.** nessa ordem.

A função **bsearch** retoma um ponteiro para um elemento para o qual se verifique a igualdade, ou um ponteiro nulo, se não for encontrado nenhum elemento para o qual a igualdade seja verificada. Se dois elementos forem iguais, o elemento que atende à igualdade não é especificado.

void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Ordena um array de **nmemb** objetos. O elemento inicial é apontado por **base.** O tamanho de cada objeto é especificado por **size.** O conteúdo do array é colocado na ordem ascendente conforme uma função de comparação apontada por **compar,** que é chamada com dois argumentos que apontam para os objetos que estão sendo comparados. A função retorna um inteiro menor que, igual a ou maior que zero, se o primeiro argumento for considerado, respectivamente, menor que, igual a ou maior que o segundo. Se houver dois elementos iguais, sua ordem no array fica indefinida.

int abs(int j);

Calcula o valor absoluto de um inteiro **j**. Se o resultado não puder ser representado, o comportamento da função fica indefinido. A função **abs** retorna o valor absoluto. ***i***

div_t div(int numer, int denom);

Calcula o quociente e o resto da divisão do numerador **numer** pelo denominador **denom.** Se a divisão não for exata, o quociente resultante é o inteiro de menor magnitude que fique próximo do quociente exato. Se o resultado não puder ser representado, o comportamento fica indefinido; caso contrário, **quot * denom + rem** será igual a **numer.** A função **div** retorna uma estrutura do tipo **div_t,** que contém tanto o quociente quanto o resto. A estrutura deve conter os seguintes membros, em qualquer ordem:

int quot; /* *quociente* ***/**
int rem; /* *resto* ***/**

long int labs(long int j);

Similar à função **abs,** exceto que o argumento e o valor retomado possuem tipo **long int.**

ldiv_t ldiv(long int numer, long int denom);

Similar à função **div,** exceto que os argumentos e os membros da estrutura retomada (que tem tipo **ldiv_t)** possuem tipo **long int.**

int mblen(const char *s, size_t n);

Se **s** não for um ponteiro nulo, a função **mblen** determina o número de bytes contido no caractere de vários bytes (multibyte) apontado por **s**. Se **s** for um ponteiro nulo, a função **mbl en** retoma um valor diferente de zero ou zero, se a codificação do caractere multibyte for ou não, respectivamente, dependente de estado. Se **s** não for um ponteiro nulo, a função **mblen** retoma 0 (se **s** apontar para o caractere nulo), ou retoma o número de bytes contido no caractere multibyte (se os próximos **n** ou menos bytes

formarem um caractere multibyte válido), ou ainda retoma — 1 (se eles não formarem um caractere multibyte válido).

int mbtowc(wchar_t *pwc, const char *s, size_t n);

Se **s** não for um ponteiro nulo, a função **mbtowc** determina o número de bytes contido no caractere de vários bytes (multibyte) apontado por **s.** Então ela determina o código para o valor do tipo **wchar_t** que corresponde àquele caractere multibyte. (O valor do código que corresponde ao caractere nulo é zero.) Se o caractere multibyte for válido e **pwc** não for um ponteiro nulo, a função **mbtowc** armazena o código no objeto apontado por **pwc.** No máximo n bytes do array apontado por **s** serão examinados.
Se **s** for um ponteiro nulo, a função **mbtowc** retorna um valor diferente de zero ou zero, se a codificação do caractere multibyte for ou não, respectivamente, dependente de estado. Se **s** não for um ponteiro nulo, a função **mbtowc** retoma 0 (se **s** apontar para o caractere nulo), ou retoma o número de bytes contido no caractere multibyte convertido (se os próximos **n** ou menos bytes formarem um caractere multibyte válido), ou ainda retoma — 1 (se eles não formarem um caractere multibyte válido). Em nenhum caso o valor retomado será maior do que n ou do que o valor da macro **MB_CUR_MAX.**

int wctomb(char *s, wchar_t wchar);

A função **wctomb** determina o número de bytes necessário para representar o caractere de vários bytes (multibyte correspondente ao código cujo valor é **wchar** (incluindo qualquer modificação no estado de deslocamento, ou shift). Ela armazena a representação do caractere multibyte no objeto array apontado por s (se **s** não for um ponteiro nulo). No máximo são armazenados **MB_CUR_MAX** caracteres. Se o valor de **wchar** for zero, a função **wctomb** é deixada no seu estado inicial de deslocamento.
Se **s** for um ponteiro nulo, a função **wctomb** retoma um valor diferente de zero ou zero, se a codificação do caractere multibyte for ou não, respectivamente, dependente de estado. Se **s** não for um ponteiro nulo, a função **wctomb** retoma — 1 se o valor de **wchar** não corresponder a um caractere multibyte válido ou retoma o número de bytes contido no caractere multibyte que corresponde ao valor de **wchar.** Em nenhum caso o valor retomado será maior do que n ou do que o valor da macro **MB_CUR_MAX.**

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

A função **mbstowcs** converte uma seqüência de caracteres multibyte, que começa no estado de deslocamento (shift) atual do array apontado por **s,** em uma seqüência de códigos correspondentes e armazena não mais do que **n** códigos no array apontado por **pwcs.** Nenhum caractere multibyte após um caractere nulo (que é convertido em um código com valor zero) será examinado ou convertido. Cada caractere multibyte é convertido como se fosse feita uma chamada à função **mbtowc,** exceto que o estado de deslocamento da função **mbtowc** não é afetado.
Não mais do que **n** elementos serão modificados no array apontado por **pwcs.** O comportamento da cópia entre objetos que se sobrepõem é indefinida. Se for encontrado um caractere multibyte inválido, a função **mbstowcs** retorna **(size_t) -1.** Caso contrário, a função **mbstowcs** retorna o número de elementos do array modificados, sem incluir um código zero de encerramento, se houver algum.

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

A função **wcstombs** converte uma seqüência de códigos, que correspondem a caracteres multibyte, do array apontado por **pwcs,** em uma seqüência de caracteres multibyte que começa no estado de deslocamento (shift) atual e armazena esses caracteres multibyte no array apontado por **s,** parando se um caractere multibyte exceder

o limite total de **n** bytes ou se um caractere nulo for armazenado. Cada código é convertido como se fosse feita uma chamada à função **wctomb,** exceto que o estado de deslocamento da função **wctomb** não é afetado.
Não mais do que **n** bytes serão modificados no array apontado por **s.** Se houver cópia entre objetos que se sobrepõem, o comportamento da função fica indefinido. Se for encontrado um código que não corresponde a um caractere multibyte válido, a função **wcstombs** retorna **(size_t) -1.** Caso contrário, a função **wcstombs** retorna o número de bytes modificados, sem incluir um caractere nulo de encerramento, se houver algum.

A.12 Manipulação de Strings <string.h>

NULm0L

Uma constante de ponteiro nulo cuja definição varia conforme a implementação.

size_t

O tipo inteiro sem sinal do resultado do operador **sizeof.**

void *memcpy(void *sl, const void *s2, size_t n);

A função **memcpy** copia **n** caracteres do objeto apontado por **s2** no objeto apontado por **sl.** Se acontecer a cópia entre objetos que se sobrepõem, o comportamento fica indefinido. A função **memcpy** retorna o valor de **sl.**

void *memmove(void *sl, const void *s2, size_t n);

A função **memmove** copia **n** caracteres do objeto apontado por **s2** no objeto apontado por **s 1.** A cópia acontece como se os **n** caracteres do objeto apontado por **s2** fossem copiados inicialmente em um array temporário de **n** caracteres que não se sobrepõem aos objetos apontados por **sl** e **s2,** e depois os **n** caracteres do array temporário são copiados no objeto apontado por **sl.** A função **memmove** retorna o valor de **sl.**

char *strcpy(char *sl, const char *s2);

A função **strcpy** copia a string apontada por **s2** (incluindo o caractere nulo de terminação) no array apontado por **sl.** Se acontecer a cópia entre objetos que se sobrepõem, o comportamento da função fica indefinido. A função **strcpy** retorna o valor de **sl.**

char *strncpy(char *sl, const char *s2, size_t n);

A função **strncpy** copia não mais do que **n** caracteres (caracteres que se situam após um caractere nulo não são copiados) do array apontado por **s2** no array apontado por **sl.** Se acontecer uma cópia entre objetos que se sobrepõem, o comportamento da função fica indefinido. Se o array apontado por **s2** for uma string com comprimento menor do que **n** caracteres, são acrescentados caracteres nulos para a cópia no array apontado por **sl,** até que sejam escritos **n** caracteres. A função **strncpy** retorna o valor de **sl.**

char *strcat(char *sl, const char *s2);

A função **strcat** acrescenta uma cópia da string apontada por **s2** (incluindo o caractere nulo de terminação) ao final da string apontada por **sl.** O caractere inicial de **s2** se sobrepõe ao caractere nulo de terminação do final de **sl.** Se ocorrer a cópia entre objetos que se sobrepõem, o comportamento fica indefinido. A função **strcat** retorna o valor de **sl.**

char *strncat(char *sl, const char *s2, size_t n);

A função **strncat** acrescenta não mais que **n** caracteres (um caractere nulo e os caracteres que o seguem não são anexados) do array apontado por **s2** ao final da string apontada por **sl.** O caractere inicial de **s2** se sobrepõe ao caractere nulo de terminação do final de **sl.** Se ocorrer a cópia entre objetos que se sobrepõem, o comportamento fica indefinido. A função **strncat** retorna o valor de **sl.**

int memcmp(const void *sl, const void *s2, size_t n);

A função **memcmp** compara os primeiros **n** caracteres do objeto apontado por **sl** aos primeiros **n** caracteres do objeto apontado por **s2.** A função **memcmp** retorna um inteiro maior que, igual a ou menor que zero, conforme o objeto apontado por **sl** seja maior que, igual a ou menor que o objeto apontado por **s2.**

int strcmp(const char *sl, const char *s2);

A função **strcmp** compara a string apontada por **sl** com a string apontada por **s2.** A função **strcmp** retorna um inteiro maior que, igual a ou menor que zero, conforme a string apontada por **sl** seja maior que, igual a ou menor que a string apontada por **s2.**

int strcoll(const char *sl, const char *s2);

A função **strcoll** compara a string apontada por **sl** com a string **s2,** ambas interpretadas como apropriadas à categoria **LCCOLLATE** do local atual. A função **strcoll** retorna um inteiro maior que, igual a ou menor que zero, conforme a string apontada por **sl** seja maior que, igual a ou menor que a string apontada por **s2,** quando ambas são interpretadas como apropriadas ao local atual.

int strncmp(const char *sl, const char *s2, size_t n);

A função **strncmp** compara não mais que **n** caracteres (caracteres situados após um caractere nulo não são comparados) do array apontado por **sl** com o array apontado por **s2.** A função **strncmp** retorna um inteiro maior que, igual a ou menor que zero, conforme o array, possivelmente com terminação nula, apontado por **sl** seja maior que, igual a ou menor que o array, possivelmente com terminação nula, apontado por **s2.**

size_t strxfrm(char *sl, const char *s2, size_t n);

A função **strxf rm** transforma a string apontada por **s2** e coloca a string resultante no array apontado por **sl.** A transformação é tal que, se a função **strcmp** for aplicada a duas strings transformadas, ela retorna um valor maior que, igual a ou menor que zero, correspondendo ao resultado da função **strcoll** aplicada às duas mesmas funções originais. Não mais que **n** caracteres são colocados no array apontado por **sl,** incluindo o caractere nulo de terminação. Se **n** for zero, será permitido a **sl** ser um ponteiro nulo. Se acontecer uma cópia entre dois objetos que se sobrepõem, o comportamento fica indefinido. A função **strxfrm** retorna o comprimento da string transformada (não incluindo o caractere nulo de terminação). Se o valor for maior ou igual a **n,** o conteúdo do array apontado por **sl** é indeterminado.

void *memchr(const void *s, int c, size_t n);

A função **memchr** localiza a primeira ocorrência de **c** (convertido em **unsigned char)** nos **n** caracteres iniciais (sendo cada um deles interpretado como **uns igned char)** do objeto apontado por **s.** A função **memchr** retorna um ponteiro para o caractere localizado, ou um ponteiro nulo se o caractere não existir no objeto.

char *strchr(const char *s, int c);

A função **strchr** localiza a primeira ocorrência de **c** (convertido em **char)** na string apontada por **s.** O caractere nulo de terminação é considerado parte da string. A função **strchr** retorna um ponteiro para o caractere localizado, ou um ponteiro nulo se o caractere não existir na string.

size_t strcspn(const char *sl, const char *s2);

A função **strcspn** calcula o comprimento do máximo segmento inicial da string

apontada por **sl** que consiste inteiramente em caracteres que não sejam da string apontada por **s2.** A função **strcspn** retorna o comprimento do segmento.

char *strpbrk(const char *sl, const char *s2);

A função **strpbrk** localiza, na string apontada por **sl,** a primeira ocorrência de qualquer caractere da string apontada por **s2.** A função **strpbrk** retorna um ponteiro para o caractere, ou um ponteiro nulo se nenhum caractere de **s2** existir em **sl.**

char *strrchr(const char *s, int c) ;

A função **strrchr** localiza, na string apontada por **s,** a última ocorrência de **c** (convertido em **char).** O caractere nulo é considerado parte da string. A função **strrchr** retorna um ponteiro para o caractere, ou um ponteiro nulo se não existir **c** na string.

size_t strspn(const char *sl, const char *s2);

A função **strspn** calcula o comprimento do máximo segmento inicial da string apontada por **sl** que consiste inteiramente em caracteres da string apontada por **s2.** A função **strspn** retorna o comprimento do segmento.

char *strstr(const char *sl, const char *s2);

A função **strstr** localiza, na string apontada por **sl,** a primeira ocorrência da seqüência de caracteres (excluindo o caractere nulo de terminação) na string apontada por **s2.** A função **strstr** retorna um ponteiro para a string localizada, ou um ponteiro nulo se a string não for encontrada. Se **s2** apontar para uma string com comprimento zero, a função retorna **sl.**

char *strtok(char *sl, const char *s2);

Uma seqüência de chamadas à função **strtok** divide a string apontada por **sl** em uma seqüência de segmentos (tokens), sendo cada um deles delimitado por um caractere da string apontada por **s2.** A primeira chamada na seqüência tem **sl** como argumento e é seguida por chamadas com um ponteiro nulo como seu primeiro argumento. A string separadora apontada por **s2** pode ser diferente de uma chamada para outra.

A primeira chamada na seqüência procura, na string apontada por **sl,** o primeiro caractere que não esteja contido na string separadora atual apontada por **s2.** Se tal caractere não for encontrado, não há segmentos na string apontada por **sl** e a função **strtok** retorna um ponteiro nulo. Se tal caractere for encontrado, ele é o início do primeiro segmento.

A função **strtok** procura então, a partir desse ponto, por um caractere que esteja contido na string separadora atual. Se tal caractere não for encontrado, o segmento atual se estende até o final da string apontada por **sl,** e as procuras subseqüentes por um segmento resultarão em um ponteiro nulo. Se tal caractere for encontrado, ele é substituído por um caractere nulo, que encerra o segmento atual. A função **strtok** salva um ponteiro para o caractere seguinte, a partir do qual iniciará a procura por um segmento.

Cada chamada subseqüente, com um ponteiro nulo como valor do primeiro argumento, inicia a procura a partir do ponteiro salvo e se comporta da maneira descrita anteriormente. A implementação se comportará como se nenhuma função da biblioteca chamasse a função **strtok.** A função **strtok** retorna um ponteiro para o primeiro caractere de um segmento, ou um ponteiro nulo, se não houver segmentos.

void *memset(void *s, int c, size_t n);

A função **memset** copia o valor de **c** (convertido em **unsigned char)** em cada um dos **n** primeiros caracteres no objeto apontado por **s.** A função **memset** retorna o valor de **s.**

char *strerror(int errnum);

A função **strerror** relaciona o número do erro em **errnum** com uma string de mensagem de erro. A implementação se comportará como se nenhuma função da biblioteca chamasse a função **strerror.** A função **strerror** retorna um ponteiro para a string, cujo conteúdo é definido pela implementação. O array apontado não será modificado pelo programa, mas pode ser sobrescrito por uma chamada subseqüente à função **strerror.**

size_t strlen(const char *s);

A função **strlen** calcula o comprimento da string apontada por **s.** A função **strlen** retorna o número de caracteres que antecedem o caractere nulo.

A. 13 Data e Hora <time.h>

CLOCKS_PER_SEC

O número por segundo correspondente ao valor retornado pela função **clock.**

NULL

Uma constante de ponteiro nulo definida pela implementação.

clock_t

Um tipo aritmético capaz de representar horas.

time_t

Um tipo aritmético capaz de representar horas.

size_t

O tipo inteiro sem sinal do resultado do operador **sizeof.**

struct tm

Contém os componentes da hora referente ao calendário, chamada *broken-down time {hora "modificada"). A* estrutura deve conter pelo menos os seguintes membros, em qualquer ordem. A semântica dos membros e seus intervalos normais são expressos nos comentários.

int tm_sec; /* *segundos após o minuto — [0,61]* ***/**
int tm_min; /* *minutos após a hora — [0, 59]* ***/**
int tm_hour; /* *horas desde a meia-noite — [0, 23]* ***/**
int tm_mday; /* *dia do mês — [1, 31]* ***/**
int tm_mon; /* *meses desde janeiro — [0,11]****/**
int tm__year; /* *anos desde 1900* ***/**
int tm_wday; /* *dias desde domingo — [0,6]* ***/**
int tm__yday; /* *dias desde 01 de janeiro — [0, 365]* ***/**
int tm_isdst; /* *sinalizador de horário de verão (Daylight Saving Time)* ***/**

O valor de **tm_isdst** é positivo se o horário de verão (Daylight Saving Time) estiver em vigência, zero se ele não estiver em vigência e negativo se a informação não estiver disponível.

clock_t clock(void);

A função **clock** determina o tempo de utilização do processador. A função **clock** retorna a melhor aproximação da implementação para o tempo do processador usado pelo programa desde o início de um período definido pela implementação relacionado apenas com a chamada do programa. Para determinar o tempo em segundos, o valor retornado pela função **clock** deve ser dividido pelo valor da macro **CLOCKS_PER_SECOND.** Se o tempo do processador usado não estiver disponível ou seu valor não puder ser representado, a função retorna o valor **(clock_t) -1.**

double difftime(time_t timel, time_t timeO);

A função **difftime** calcula a diferença entre duas horas do calendário: **timel - time0.** A função **difftime** retorna a diferença expressa em segundos como **double.**

time_t mktime(struct tm *timeptr);

A função **mktime** converte a hora "modificada" (broken-down time) na estrutura apontada por **timeptr,** expressa como hora local, em um valor de hora de calendário com a mesma codificação dos valores retornados pela função **time.** Os valores originais dos componentes **tm_wday** e **tm_yday** da estrutura são ignorados, e os valores originais dos outros componentes não são restritos aos intervalos indicados anteriormente. Após um encerramento bem-sucedido, os valores dos componentes **tm_wday** e **tm vdav** da estrutura são definidos apropriadamente e os outros componentes são definidos para representar a hora de calendário especificada, mas com seus valores obrigados a estarem dentro dos intervalos indicados anteriormente; o valor final de **tm_mday** não é definido até que **tm_mon** e **tm_year** sejam determinados. A função **mktime** retorna a hora de calendário especificada codificada como um valor do tipo **t ime_t**. Se a hora de calendário não puder ser representada, a função retorna o valor **(time_t) -1.**

time_t time(time_t *timer);

A função **time** determina a hora atual de calendário. A função **time** retorna a melhor aproximação da implementação para a hora atual de calendário. O valor **(time_t) -1** é retornado se a hora de calendário não estiver disponível. Se **timer** não for um ponteiro nulo, o valor de retorno também é atribuído ao objeto para o qual esse ponteiro aponta.

char *asctime(const struct tm *timeptr);

A função **asctime** converte a hora "modificada" (broken-down time), existente na estrutura apontada por **timeptr,** em uma string da forma
Sun Sep 16 01:03:52 1973\n\0 A função **asctime** retorna um ponteiro para a string.

char *ctime(const time_t *timer);

A função **ctime** converte a hora de calendário apontada por **timer** em hora local na forma de uma string. Ela é equivalente a
asctime(localtime(timer)) A função **ctime** retorna o ponteiro retornado por **asctime** com aquela hora "modificada" como argumento.

struct tm *gmtime(const time_t *timer);

A função **gmtime** converte a hora de calendário apontada por **timer** em uma hora "modificada", expressa como UTC (Coordinated Universal Time). A função **gmtime** retorna um ponteiro para aquele objeto, ou um ponteiro nulo se UTC não estiver disponível.

struct tm *localtime(const time_t *timer);

A função **localtime** converte a hora de calendário apontada por **timer** em hora "modificada", expressa como hora local. A função **localtime** retorna um ponteiro para aquele objeto.

size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);

A função **strftime** coloca caracteres no array apontado por **s** da maneira controlada pela string apontada por **format.** A string **f ormat** consiste em zero ou mais especificadores de conversão e caracteres multibyte comuns. Todos os caracteres comuns (incluindo o caractere nulo de terminação) são copiados inalterados no array. Se acontecer cópia entre objetos que se sobrepõem, o comportamento da função fica

indefinido. Não mais que **maxsize** caracteres são colocados no array. Cada especificador de conversão é substituído pelos caracteres apropriados conforme a descrição na lista a seguir. Os caracteres apropriados são determinados pela categoria **LC_TIME** do local atual e pelos valores contidos na estrutura apontada por **timeptr.**
%a é substituído pelo nome abreviado do dia da semana do local.
%A é substituído pelo nome completo do dia da semana do local.
%b *é* substituído pelo nome abreviado do mês do local.
%B é substituído pelo nome completo do mês do local.
%c é substituído pela representação apropriada da data e hora do local.
%d *é* substituído pelo dia do mês como número decimal **(01-31).**
%H é substituído pela hora (relógio de 24 horas) como número decimal **(00-23).**
%I é substituído pela hora (relógio de 12 horas) como número decimal **(01-12).**
%j é substituído pelo dia do ano como número decimal **(001-366).**
%m é substituído pelo mês como número decimal **(01-12).**
%M. é substituído pelo minuto como número decimal **(00-59).**
%g é substituído pelo equivalente do local às designações AM/PM associadas ao relógio de 12 horas.
%S *é* substituído pelo segundo como número decimal **(00-61).**
%U é substituído pelo número da semana no ano (tendo o primeiro domingo como o primeiro dia da semana 1) como número decimal **(00-53).**
%w é substituído pelo dia da semana como número decimal **(0-6),** onde o domingo é **0.**
%W é substituído pelo número da semana no ano (tendo a primeira segunda-feira como o primeiro dia da semana 1) como número decimal **(00-53).**
%x é substituído pela representação apropriada da data do locai.
%X é substituído pela representação apropriada da hora do local.
%y é substituído pelo ano, sem o século, como número decimal **(00-99).**
%Y é substituído pelo ano, com o século, como número decimal.
%Z *é* substituído pelo nome ou abreviação do fuso horário, ou por nenhum caractere, se nenhum fuso horário puder ser determinado.
%% *é* substituído por %.

Se um especificador de conversão não for igual aos mencionados anteriormente, o comportamento fica indefinido. Se o número total de caracteres resultantes incluindo o caractere nulo de terminação não for maior do que **maxsize,** a função **strftime** retorna o número de caracteres colocados no array apontado por **s,** não incluindo o caractere nulo de terminação. Caso contrário, é retornado zero e o conteúdo do array fica indeterminado.

A.14 Limites de Implementação

<limits.h>

As constantes a seguir devem ser definidas de forma a serem iguais ou maiores, em valor absoluto (magnitude), que os valores indicados.

#define CHAR_BIT 8

O número de bits do menor objeto que não seja um campo de bit (byte).

#define SCHAR__MIN -127

O valor mínimo de um objeto do tipo **signed char.**

#define SCHARMAX +127

O valor máximo de um objeto do tipo **signed char.**

#define UCHAR_MAX 255

O valor máximo de um objeto do tipo **unsigned char.**

#define CHAR_MIN 0 ou SCHAR_MIN

O valor mínimo de um objeto do tipo **char.**

#define CHAR_MAX UCHAR_MAX OU SCHAR_MAX

O valor máximo de um objeto do tipo **char.**

#define MB_LEN_MAX 1

O número máximo de bytes em um caractere multibyte, para qualquer local suportado.

#define SHRT_MIN -32767

O valor mínimo de um objeto do tipo **short int.**

#define SHRT_MAX +32767

O valor máximo de um objeto do tipo **short int.**

#define USHRT_MAX 65535

O valor máximo de um objeto do tipo **unsigned short int.**

#define INT_MIN -32767

O valor mínimo de um objeto do tipo **int.**

#define INT_MAX +32767

O valor máximo de um objeto do tipo **int.**

#define UINT_MAX 65535

O valor máximo de um objeto do tipo **unsigned int.**

#define LONG_MIN -2147483647

O valor mínimo de um objeto do tipo **long int.**

#define LONG_MAX +2147483647

O valor máximo de um objeto do tipo **long int.**

#define ULONG_MAX 4294967295
O valor máximo de um objeto do tipo **unsigned long int.**

<float.h>

#define FLTROUNDS
O modo de arredondamento para soma de valores de pontos flutuantes.
-1 indeterminável **0** para zero
1 para o mais próximo
2 para o infinito positivo (mais infinito)
3 para o infinito negativo (menos infinito)

As constantes a seguir devem ser definidas de forma a serem iguais ou maiores, em valor absoluto, aos valores indicados.

#define FLT_RADIX 2
A base de uma representação exponencial, *b*.

#define FLT_MANT_DIG
#define LDBL_MANT_DIG
#define DBL_MANT_DIG
O número de dígitos da base **FLT_RADIX,** *p*, contidos no conjunto total em ponto flutuante de algarismos significativos.

#define FLT_DIG 6
#define DBL_DIG 10
#define LDBL_DIG 10

O número de dígitos decimais, *q*, tal que qualquer número de ponto flutuante com *q* dígitos decimais possa ser arredondado para um número de ponto flutuante com *p* dígitos na base *b* e retomar sem modificação nos *q* dígitos decimais.
#define FLT_MIN_EXP #define DBL_MIN_EXP #define LDBL_MIN_EXP

O mínimo inteiro negativo tal que **FLT_RADIX** elevado àquela potência menos 1 seja um número de ponto flutuante normalizado.

#define FLT_MIN_10_EXP -37
#define DBL_MIN_10_EXP -37
#define LDBL_MIN_10_EXP -37

O mínimo inteiro negativo tal que 10 elevado àquela potência esteja no intervalo dos números de ponto flutuante normalizados.

#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP

O máximo inteiro tal que **FLT_RADIX** elevado àquela potência menos 1 seja um número de ponto flutuante finito representavel.

#define FLT_MAX_10_EXP +37
#define DBL_MAX_10_EXP +37
#define LDBL_MAX_10_EXP +37

O máximo inteiro tal que 10 elevado àquela potência esteja no intervalo dos números de ponto flutuante finitos representáveis.

As constantes a seguir devem ser definidas de forma a serem iguais ou maiores que os valores indicados.

#define FLT_MAX 1E+37
#define DBL_MAX 1E+37
#define LDBLMAX 1E+37

O máximo número de ponto flutuante finito representável.

As constantes a seguir devem ser definidas de forma a serem iguais ou menores que os valores indicados.

#define FLT_EPSILON 1E-5
#define DBL_EPSILON 1E-9
#define LDBL_EPSILON 1E-9

A diferença entre 1.0 e o menor número maior que 1.0 que seja representável no tipo de ponto flutuante **deter**minado.

#define FLT_MIN 1E-37
#define DBL_MIN 1E-37
#define LDBL_MIN 1Z-3"

O número de ponto flutuante mínimo normalizado positivo.

Apêndice B
Precedência de Operadores e Associatividade

<table>
<tr><th>Operador</th><th>Associatividade</th></tr>
<tr><td>( )   [ ]   ->   •</td><td>esquerda para a direita</td></tr>
<tr><td>+ +   —   +   -   !   ~   (tipo)   *   & sizeof</td><td>direita para a esquerda</td></tr>
<tr><td>*   /   %</td><td>esquerda para a direita</td></tr>
<tr><td>+   -</td><td>esquerda para a direita</td></tr>
<tr><td>« »</td><td>esquerda para a direita</td></tr>
<tr><td><   <=   > >=</td><td>esquerda para a direita</td></tr>
<tr><td>==   !=</td><td>esquerda para a direita</td></tr>
<tr><td>&</td><td>esquerda para a direita</td></tr>
<tr><td>^</td><td>esquerda para a direita</td></tr>
<tr><td>|</td><td>esquerda para a direita</td></tr>
<tr><td>&&</td><td>esquerda para a direita</td></tr>
<tr><td>| |</td><td>esquerda para a direita</td></tr>
<tr><td>?:</td><td>direita para a esquerda</td></tr>
<tr><td>=   +=   -=   *=   /=   %=   &=   ^= |=   <<=   >>=</td><td>direita para a esquerda</td></tr>
<tr><td>`</td><td>esquerda para a direita</td></tr>
</table>

Os operadores estão mostrados na ordem decrescente de precedência, de cima para baixo.

Apêndice C

Conjunto de Caracteres ASCII

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	die	del	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	**#**	**$**	**%**	&	`
4	(	)	*	+	,	-		**/**	0	1
5	2	3	4	5	6	7	8	9	:	**;**
6	<	=	>	?	**@**	A	B	C	D	E
7	F	G	H	**I**	J	K	L	M	N	O
8	**P**	Q	R	S	T	U	V	W	X	Y
9	Z	**[**	\	**]**	**A**	_	'	a	b	c
10	d	e	f	g	h	i	j	k	1	m
11	n	0	P	q	r	s	t	u	**V**	w
12	x	y	z	{	|	}	~	del		

Os dígitos à esquerda da tabela são os dígitos da esquerda do equivalente decimal (0-127) do código do caractere e os dígitos do topo da tabela são os dígitos da direita do código do caractere. Por exemplo, o código do caractere 'F' é 70 e o código do caractere '&' é 38.

Apêndice D
Sistemas de Numeração

Objetivos

• Entender os conceitos básicos dos sistemas de numeração tais como base, valor posicional e valor do símbolo.

• Entender como trabalhar com números representados nos sistemas de numeração binário, octal e hexadecimal.

• Ser capaz de exprimir números binários como números octais ou hexadecimais.

• Ser capaz de converter números octais e hexadecimais em números binários.

• Ser capaz de fazer conversões de números decimais para seus equivalentes binários, octais e hexadecimais, e vice-versa.

• Entender a aritmética binária e como números binários negativos são representados usando a notação de complemento de dois.

Aqui só há números ratificados.
William Shakespeare

A natureza possui algum tipo de sistema de coordenadas aritmético-geométricas, porque possui todos os tipos de modelos. O que experimentamos da natureza está em modelos e todos os modelos da natu reza são extremamente bonitos. Ocorre-me que o sistema da natureza deve ser uma beleza autêntica, porque em química achamos que todas as associações sempre estão em belos números inteiros — não há frações.
Richard Buckminster Fuller

Sumário

D.1 Introdução

Neste apêndice, apresentamos os sistemas principais de numeração que os programadores em linguagem C utilizam, especialmente quando estão trabalhando em projetos de software que exigem muita interação com hardware em "nível de máquina". Projetos como esses incluem sistemas operacionais, software de redes de computadores, compiladores, sistemas de bancos de dados e aplicações que exigem alto desempenho.

Quando escrevemos um inteiro como 227 ou —63 em um programa em C, assume-se que o número está no *sistema de numeração decimal (base 10).* Os *dígitos* no sistema de numeração decimal são 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. O menor dígito é 0 e o maior dígito é 9 — um a menos que a *base* 10. Internamente, os computadores usam o s*istema de numeração binário (base 2). O* sistema de numeração binário tem apenas dois dígitos, que são 0 e 1. Seu menor dígito é 0 e seu maior dígito é 1 — um a menos que a base 2.

Como veremos, os números binários tendem a ser muito maiores do que seus equivalentes decimais. Os programadores que trabalham com linguagens assembly e com linguagens de alto nível como o C, que permitem que eles desçam ao "nível de máquina", acham complicado trabalhar com números binários. Assim, dois outros sistemas de numeração — *o sistema de numeração octal (base 8)* e o *sistema de numeração hexadecimal (base 16)* — são populares principalmente porque tornam conveniente a abreviação de números binários.

No sistema de numeração octal, os dígitos variam de 0 a 7. Em vista de tanto o sistema de numeração binário quanto o sistema octal possuírem menos dígitos que o sistema decimal, seus dígitos são os mesmos que seus correspondentes do sistema decimal.

O sistema hexadecimal apresenta um problema porque ele exige dezesseis dígitos — o menor dígito é 0 e o maior dígito tem valor equivalente a 15 (um a menos que a base 16). Por convenção, usamos as letras de A a F para representar os dígitos correspondentes aos valores decimais de 10 a 15. Desta forma, em hexadecimal podemos ter números como 876 consistindo apenas em dígitos semelhantes aos decimais, números como 8A55F consistindo em dígitos e letras e números como FFE consistindo apenas em letras. Ocasionalmente, um número hexadecimal é grafado como uma palavra comum como FACE ou FADA — isso pode parecer estranho aos programadores acostumados a trabalhar com números.

Cada um desses sistemas de numeração usa a *notação posicionai* — cada posição, na qual é escrito um dígito, possui um *valor posicionai* diferente. Por exemplo, no número decimal 937 (o 9, o 3 e o 7 são chamados *valores dos símbolos* ou *valores dos algarismos),* dizemos que o 7 é escrito na *posição das unidades,* o 3 é escrito na *posição das dezenas* e o 9 é escrito na *posição das centenas.* Observe que cada uma dessas posições é uma potência da base (base 10) e que essas potências começam em 0 e aumentam de 1 à medida que nos movemos para a esquerda do número (Fig. D.3).

Para números decimais maiores, as próximas posições à esquerda seriam a *posição dos milhares* (10 elevado à terceira potência), a *posição das dezenas de milhar* (10 elevado à quarta potência), a *posição das centenas de milhar* (10 elevado à quinta potência), a *posição dos milhões* (10 elevado à sexta potência), a *posição das dezenas de milhão* (10 elevado à sétima potência) e assim por diante.

No número binário 101, dizemos que o 1 da extremidade da direita está escrito na *posição do um, o* 0 está escrito na *posição do dois* e o 1 da extremidade esquerda está escrito na *posição do quatro.* Veja que cada uma dessas posições é uma potência da base (base 2) e que essas potências começam em 0 e aumentam de 1 à medida que nos movemos para a esquerda do número (Fig. D.4).

Para números binários mais longos, as próximas posições à esquerda seriam a *posição do oito* (2 elevado à terceira potência), a *posição do dezesseis* (2 elevado à quarta potência), a *posição do trinta e dois* (2 elevado à quinta potência), a *posição do sessenta e quatro* (2 elevado à sexta potência) e assim por diante.

No número octal 425, dizemos que o 5 está escrito na *posição do um*, o 2 está escrito na *posição do oito* e o 4 está escrito na *posição do sessenta e quatro.* Veja que cada uma dessas posições é uma potência da base (base 8) e que essas potências começam em 0 e aumentam de 1 à medida que nos movemos para a esquerda do número (Fig. D.5).

Dígito binário	Dígito octal	Dígito Decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor decimal de 10)
			B (valor decimal de 11)
			C (valor decimal de 12)
			D (valor decimal de 13)
			E (valor decimal de 14)
			F (valor decimal de 15)

Fig. D.l Dígitos dos sistemas de numeração binário, octal, decimal e hexadecimal,

Atributo	Dígito binário	Dígito octal	Dígito Decimal	Dígito hexadecimal
Base	2	8	10	16
Menor dígito	0	0	0	0
Maior dígito	1	7	9	F

Fig. D.2 Comparação dos sistemas de numeração binário, octal, decimal e hexadecimal.

Valores posicionais no sistema de numeração decimal			
Dígito decimal	9	3	7
Nome da posição	Centenas	Dezenas	Unidades
Valor posicional	100	10	1
Valor posicional como uma potencia na base (10)	10^2	10^1	10^0

Fig. D.3 Valores posicionais no sistema de numeração decimal.

Valores posicionais no sistema de numeração binário			
Dígito Binário	1	0	1
Nome da posição	Quatro	Dois	Um
Valor posicional	4	2	1
Valor posicional como uma potencia na base (2)	2^2	2^1	2^0

Fig. D.4 Valores posicionais no sistema de numeração binário.

Valores posicionais no sistema de numeração octal			
Dígito octal	4	2	5
Nome da posição	Sessenta e qutro	Oito	Um
Valor posicional	64	8	1
Valor posicional como uma potencia na base (8)	8^2	8^1	8^0

Fig. D.5 Valores posicionais no sistema de numeração octal.

Valores posicionais no sistema de numeração hexadecimal			
Dígito hexadecimal	3	D	A
Nome da posição	Duzentos e cinqüenta e seis	Dezesseis	Um
Valor posicional	256	16	1
Valor posicional como uma potencia na base (16)	16^2	16^1	16^0

Fig. D.6 Valores posicionais no sistema de numeração hexadecimal.

Para números octais mais longos, as próximas posições à esquerda seriam a *posição do quinhentos e doze* (8 elevado à terceira potência), a *posição do quatro mil e noventa e seis* (8 elevado à quarta potência), a *posição do trinta e dois mil, setecentos e sessenta e oito* (8 elevado à quinta potência) e assim por diante.

No número hexadecimal 3DA, dizemos que o A está escrito na *posição do um*, o D está escrito na *posição do dezesseis* e o 3 está na *posição do duzentos e cinqüenta e seis*. Veja que cada uma dessas posições é uma potência da base (base 16) e que essas potências começam em 0 e aumentam de 1 à medida que nos movemos para a esquerda do número (Fig. D.6).

Para números hexadecimais mais longos, as próximas posições seriam a *posição*

do quatro mil e noventa e seis (16 elevado à terceira potência), a *posição do trinta e dois mil, setecentos e sessenta e oito* (16 elevado à quarta potência) e assim por diante.

D.2 Exprimindo Números Binários como Números Octais e Hexadecimais

O uso principal dos números octais e hexadecimais em computação é para exprimir representações binárias longas. A Fig. D.7 destaca o fato de que números binários longos podem ser expressos de uma forma concisa em sistemas de numeração com bases maiores do que a do sistema de numeração binário.

Um relacionamento particularmente importante com o sistema de numeração binário, que tanto o sistema de numeração octal quanto o hexadecimal possuem, é que suas bases (8 e 16, respectivamente) são potências da base do sistema de numeração binário (base 2). Examine o número binário com 12 dígitos a seguir e seus equivalentes octal e hexadecimal. Veja se você pode determinar como esse relacionamento torna conveniente exprimir números binários em números octais e hexadecimais. A resposta vem após os números.

Número Binário	Equivalente Octal	Equivalente Hexadecimal
100011010001	**4321**	**8D1**

Para ver como o número binário é convertido facilmente em um número octal, simplesmente divida o número binário de 12 dígitos em grupos de três bits consecutivos cada um e escreva aqueles grupos sobre os dígitos correspondentes do número octal, como se segue

100	011	010	001
4	3	2	1

Observe que o dígito octal escrito sob cada grupo de três bits corresponde exatamente ao equivalente octal daquele número binário de 3 dígitos, de acordo com o que mostra a Fig. D.7.

O mesmo tipo de relacionamento pode ser observado na conversão de números do sistema binário para o hexadecimal. Em particular, divida o número binário de 12 dígitos em grupos de quatro bits consecutivos cada um e escreva aqueles grupos sobre os dígitos correspondentes do número hexadecimal, como se segue

1000	1101	0001
8	D	1

Observe que o dígito hexadecimal escrito sob cada grupo de quatro bits corresponde exatamente ao equivalente hexadecimal daquele número binário de quatro dígitos, como mostra a Fig. D.7.

Número decimal	Representação binária	Representação octal	Representação hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. D.7 Equivalentes decimais, binários, octais e hexadecimais.

Convertendo um número binário em decimal						
Valores posicionais:	32	16	8	4	2	1
Valores dos algarismos	1	1	0	1	0	1
Produtos	1*32=32	1*16=16	0*8=0	4*4=4	0*2=0	1*1=1
Soma:	= 32	+ 16	+ 0	+ 4	+ 0	+1 =53

Fig. D.8 Convertendo um número binário em decimal.

D.3 Convertendo Números Octais e Hexadecimais em Números Binários

Na seção anterior, vimos como converter números binários em seus equivalentes octais e hexadecimais, formando grupos de dígitos binários e simplesmente reescrevendo esses grupos como seus valores octais e hexadecimais equivalentes. Esse processo pode ser usado na ordem inversa para produzir o número binário equivalente a um número octal ou hexadecimal dado.

Por exemplo, o número octal 653 é convertido em binário simplesmente escrevendo o 6 como seu número binário equivalente de 3 dígitos 110, o 5 como seu binário de 3 dígitos equivalente 101 e o 3 como seu binário de 3 dígitos equivalente 011 para formar o número binário de 9 dígitos 110101011.

O número hexadecimal FAD5 é convertido em binário simplesmente escrevendo o F como seu número binário equivalente de 4 dígitos 1111, o A como seu binário de 4 dígitos equivalente 1010, o D como seu binário de 4 dígitos equivalente 1101 e o 5 como seu binário de 4 dígitos equivalente 0101 para formar o número binário de 16 dígitos 1111101011010101.

D.4 Convertendo do Sistema Binário, Octal ou Hexadecimal para o Sistema Decimal

Por estarmos acostumados a trabalhar com decimais, freqüentemente é útil converter um número binário, octal ou hexadecimal em decimal, para ter uma noção do que o número "realmente" vale. Nossos diagramas na Seção D.1 expressam os valores posicionais em decimais. Para converter um número em decimal, a partir de outra base, multiplique o equivalente decimal de cada dígito por seu valor posicionai e some esses produtos. Por exemplo, o número binário 110101 é convertido no decimal 53 de acordo com o que mostra a Fig. D.8.

Para converter o octal 7614 no decimal 3980, aplicamos a mesma técnica, usando dessa vez os valores posicionais octais apropriados, como mostra a Fig. D.9.

Para converter o hexadecimal AD3B no decimal 44347, aplicamos a mesma técnica, usando dessa vez os valores posicionais hexadecimais apropriados, como mostra a Fig. D. 10.

D.5 Convertendo do Sistema Decimal para o Sistema Binário, Octal ou Hexadecimal

As conversões das seções anteriores são conseqüências naturais das convenções da notação posicionai. Converter do sistema decimal para o sistema binário, octal ou hexadecimal também segue essas convenções.

Suponha que desejamos converter o número decimal 57 para o sistema binário. Começamos escrevendo os valores posicionais das colunas, da direita para a esquerda, até alcançarmos a coluna cujo valor posicionai seja maior do que o número decimal. Não precisamos daquela coluna, portanto a descartamos. Assim, escrevemos inicialmente:

Valores posicionais: **64 32 16 8 4 2 1**
A seguir descartamos a coluna com valor 64, restando:
Valores posicionais: **32 16 8 4 2 1**

Agora trabalhamos a partir da coluna da extremidade esquerda em direção à direita. Dividimos 57 por 32 e observamos que há uma vez 32 em 57, com resto 25, portanto escrevemos 1 na coluna 32. Dividimos 25 por 16 e observamos que há uma vez 16 em 25, com resto 9 e escrevemos 1 na coluna 16. Dividimos 9 por 8 e observamos

Convertendo um número octal em decimal				
Valores posicionais:	512	64	8	1
Valores dos algarismos	7	6	1	4
Produtos	7*512=3584	6*64 = 384	1*8=8	1*4 = 4
Soma:	= 3584 + 3384	+ 8 + 4 =	3980	

Fig. D.9 Convertendo um número octal em decimal.

Convertendo um número hexadecimal em decimal				
Valores posicionais:	4096	256	16	1
Valores dos algarismos	A	D	3	4
Produtos	A*4096=40960	D*256=3328	3*16=48	B*1=11
Soma:	= 40960+ 3328	+ 48 + 11 =	44347	

Fig. D. 10 Convertendo um número hexadecimal em decimal.

que há uma vez 8 em 9, com resto 1. As duas próximas colunas produzem quocientes zero quando 1 é dividido por seus valores posicionais, portanto escrevemos Os nas colunas 4 e 2. Finalmente, dividindo 1 por 1 obtemos 1, portanto escrevemos 1 na coluna 1. Isso leva a:

Valores posicionais: **32 16 8 4 2 1**
Valores dos algarismos: **1110 0 1**

e assim o valor decimal 57 é equivalente ao binário 111001.

Para converter o número decimal 103 para o sistema octal, começamos escrevendo os valores das colunas até alcançarmos a coluna cujo valor posicionai seja maior do que o número decimal. Não precisamos daquela coluna, portanto a descartamos. Assim, escrevemos inicialmente:

Valores posicionais: **512 64 8 1**
A seguir descartamos a coluna com valor 512, restando:
Valores posicionais: **64 8 1**

Agora trabalhamos a partir da coluna da extremidade esquerda em direção à direita. Dividimos 103 por 64 e observamos que há uma vez 64 em 103, com resto 39, portanto escrevemos 1 na coluna 64. Dividimos 39 por 8 e observamos que há quatro vezes 8 em 39, com resto 7 e escrevemos 4 na coluna 8. Finalmente, dividimos 7 por 1 e observamos que há sete vezes 1 em 7, não ficando resto algum, portanto escrevemos 7 na coluna 1. Isso leva a:

Valores posicionais: **64 8 1**
Valores dos algarismos: **14 7**
e assim o valor decimal 103 é equivalente ao octal 147.

Para converter o número decimal 375 para o sistema hexadecimal, começamos escrevendo os valores das colunas até alcançarmos a coluna cujo valor posicionai seja maior do que o número decimal. Não precisamos daquela coluna, portanto a descartamos. Assim, escrevemos inicialmente:

Valores posicionais: **4096 256 16 1**
A seguir descartamos a coluna com valor 4096, restando:
Valores posicionais: **256 16 1**

Agora trabalhamos a partir da coluna da extremidade esquerda em direção à direita. Dividimos 375 por 256 e observamos que há uma vez 256 em 375, com resto 119, portanto escrevemos 1 na coluna 256. Dividimos 119 por 16 e observamos que há sete vezes 16 em 119, com resto 7 e escrevemos 7 na coluna 16. Finalmente, dividimos 7 por 1 e observamos que há sete vezes 1 em 7, não ficando resto algum, portanto escrevemos 7 na coluna 1. Isso leva a:

Valores posicionais: **256 16 1**
Valores dos algarismos: **17 7**
e assim o valor decimal 375 é equivalente ao hexadecimal 177.

D.6 Números Binários Negativos: Notação de Complemento de Dois

A análise deste apêndice concentrou-se nos números positivos. Nesta seção, explicamos como os computadores representam números negativos usando a *notação de complemento de dois.* Em primeiro lugar explicamos como é formado o complemento de dois de um número binário e depois mostramos por que ele representa o valor negativo de um determinado número binário.

Considere um equipamento com inteiros de 32 bits. Suponha

int valor = 13;

A representação em 32 bits de **valor** é

00000000 00000000 00000000 00001101

Para formar o negativo de **valor,** formamos inicialmente o complemento de um, aplicando o operador de complemento bit a bit do C, (~):

complemento_um_de_valor = -valor;

Internamente, **-valor** é agora **valor** com cada um de seus bits invertidos — os uns se tornam zeros e os zeros se tornam uns, como se segue:

valor:
00000000 00000000 00000000 00001101

-valor (i.e., complemento de um de **valor):**

11111111 11111111 11111111 11110010

Para formar o complemento de dois de **valor,** simplesmente adicionamos um ao complemento de um de **valor.** Desta forma

O complemento de dois de **valor:**

11111111 11111111 11111111 11110011

Agora se isso é realmente igual a —13, devemos ser capazes de adicionar o binário 13 a ele e obter o resultado 0. Vamos tentar fazer isso:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
 -----------------------------------
 00000000 00000000 00000000 00000000
```

O bit transportado da coluna da extremidade esquerda é descartado e realmente obtemos zero como resultado. Se adicionarmos o complemento de um de um número ao próprio número, o resultado será todo ls. O segredo de obter um resultado todo em zeros é que o complemento de dois vale 1 a mais do que o complemento de um. A adição de 1 faz com que cada coluna resulte em zero, transportando o valor 1 para a próxima coluna. O valor é transportado para a esquerda, de uma coluna para outra até que seja descartado do bit da extremidade esquerda e assim o número resultante fica todo consistindo em zeros.

Na realidade, os computadores realizam uma subtração como

x = a — valor;

adicionando o complemento de dois de **valor** a **a** como se segue:

x = a + (-valor + 1);

Suponha que **a** é 27 e **valor** é 13, como antes. Se o complemento de dois de **valor** é realmente o negativo de **valor,** adicionar a **a** deve produzir o resultado 14. Vamos tentar fazer isso:

a (i.e.,27)	**00000000 00000000 00000000 00011011**
+(-valor +1)	**+ 11111111 11111111 11111111 11110011**
	00000000 00000000 00000000 00001110

que é realmente igual a 14.

Resumo

• Quando escrevemos um inteiro como 19 ou 227 ou —63 em um programa C, o número é considerado automaticamente como estando no sistema de numeração decimal (base 10). Os dígitos no sistema de numeração decimal são 0, 1,2, 3, 4, 5, 6, 7, 8 e 9. O menor dígito é 0 e o maior dígito é 9 — um a menos que a base 10.

• Internamente, os computadores usam o sistema de numeração binário (base 2). O sistema de numeração binário tem apenas dois dígitos, que são 0 e 1. Seu menor dígito é 0 e seu maior dígito é 1 — um a menos que a base 2.

• O sistema de numeração octal (base 8) e o sistema de numeração hexadecimal (base 16) se tornaram populares principalmente porque são convenientes para exprimir números binários de uma forma abreviada.

• Os dígitos do sistema de numeração octal variam de 0 a 7.

• O sistema de numeração hexadecimal apresenta um problema porque exige dezesseis dígitos — o menor dígito com valor 0 e o maior dígito com um valor equivalente a 15 decimal (um a menos que a base 16). Por convenção usamos as letras A a F para representar os dígitos hexadecimais correspondentes aos valores decimais 10 a 15.

• Cada sistema de numeração usa notação posicionai — cada posição na qual um dígito é escrito tem um valor posicional diferente.

• Um relacionamento importante que tanto o sistema de numeração octal quanto o sistema de numeração hexadecimal possuem com o sistema binário é que as bases dos sistemas octal e hexadecimal (8 e 16, respectivamente) são potências da base do sistema de numeração binário (base 2).

• Para converter um número octal em um número binário, simplesmente substitua cada dígito octal pelo binário equivalente de três dígitos.

• Para converter um número hexadecimal em um número binário, simplesmente substitua cada dígito hexadecimal pelo binário equivalente de quatro dígitos.

• Por estarmos acostumados a trabalhar com números decimais, freqüentemente é útil converter um número binário, octal ou hexadecimal em decimal para ter uma melhor noção do que o número "realmente" vale.

• Para converter um número de outra base para um número decimal, multiplique o equivalente decimal de cada dígito por seu valor posicionai e some esses produtos.

• Os computadores representam números negativos usando a notação de complemento de dois.

• Para formar o negativo de um valor, forme inicialmente seu complemento de um aplicando o operador de complemento bit a bit do C (~). Isso inverte os bits do valor. Para formar o complemento de dois de um valor, simplesmente adicione um ao complemento de um do valor.

Terminologia

base
conversões
dígito
notação de complemento de dois
notação de complemento de um
notação posicional
operador de complemento bit a bit (~)
sistema de numeração base 2
sistema de numeração base 8
sistema de numeração base 10
sistema de numeração base 16
sistema de numeração binário
sistema de numeração decimal
sistema de numeração hexadecimal
sistema de numeração octal
valor do algarismo
valor do símbolo
valor negativo
valor posicional

Exercícios de Revisão

D.1 As bases dos sistemas de numeração decimal, binário, octal e hexadecimal são ____________, __________________ e ______________.

D.2 Em geral, as representações decimal, octal e hexadecimal de um determinado número binário contêm (mais/menos) dígitos do que o número binário.

D.3 (Verdadeiro/Falso) Um motivo popular para o uso do sistema de numeração decimal é que ele forma uma notação conveniente para exprimir números binários de uma forma abreviada, simplesmente substituindo um dígito decimal por um grupo de quatro dígitos binários.

D.4 A representação (octal/hexadecimal/decimal) de um valor binário muito grande é a mais concisa (das alternativas fornecidas).

D.5 (Verdadeiro/Falso) O maior dígito em qualquer base vale um a mais que a base.

D.6 (Verdadeiro/Falso) O menor dígito em qualquer base vale um a menos que a base.

D.7 O valor posicional do dígito à esquerda do dígito da extremidade direita nos sistemas binário, octal, decimal ou hexadecimal é sempre ____________________.

D.8 Preencha as lacunas na tabela a seguir com os valores posicionais das quatro posições da direita em cada um dos sistemas de numeração indicados.

decimal	**1000**	**100**	**10**	**1**
hexadecimal	...	**256**	**...**	**...**
binário	...	...	...	**...**
octal	**512**	**...**	**8**	**...**

D.9 Converta o binário **110101011000** para os sistemas octal e hexadecimal.

D.10 Converta o hexadecimal **FACE** para o sistema binário.

D.11 Converta o octal **7316** para o sistema binário.

D.12 Converta o hexadecimal **4FEC** para o sistema octal. (Sugestão: Em primeiro lugar converta **4FEC** para o sistema binário e depois converta aquele número binário para o sistema octal.)

D.13 Converta o binário **1101110** para o sistema decimal.

D.14 Converta o octal **317** para o sistema decimal.

D.15 Converta o hexadecimal **EFD4** para o sistema decimal.

D.16 Converta o decimal **177** para os sistemas binário, octal e hexadecimal.

D.17 Mostre a representação **binária** do decimal **417.** Depois mostre os complementos de um e de dois de **417.**

D.18 Qual o resultado quando o complemento de um de um número é adicionado ao próprio número?

Respostas dos Exercícios de Revisão

D.1 **10, 2, 8, 16.**

D.2 Menos.

D.3 Falso.

D.4 Hexadecimal.

D.5 Falso — O maior dígito em qualquer base vale um a menos que a base.

D.6 Falso — O menor dígito em qualquer base é zero.

D.7 **1** (a base elevada à potência zero).

D.8 A base do sistema de numeração.

D.9 Preencha as lacunas na tabela a seguir com os valores posicionais das quatro posições da direita em cada um dos sistemas de numeração indicados.

decimal **1000 100 10 1**
hexadecimal **4096 256 16 1**
binário **8 4 2 1**
octal **512 64 8 1**

D.10 Octal **6530;** Hexadecimal **D58.**

D.11 Binário **1111 1010 1100 1110.**

D.12 Binário **111 011 001 110.**

D.13 Binário **0 100 111 111 101 100;** Octal **47754.**

D.14 Decimal **2 + 4 + 8 + 32 + 64 = 110.**

D.15 Decimal **7 + 1*8 + 3*64=7 + 8 + 192 = 207.**

D.16 Decimal **4 + 13*16 + 15*256 + 14*4096 = 61396.**

D.17 Decimal **177**

para binário:
256 128 64 32 16 8 4 2 1 128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1) 10110001
para octal:

512 64 8 1 64 8 1
(2*64)+(6*8)+(1*1) 261
para hexadecimal: **256 16 1 16 1**
(11*16)+(1*1)
(B*16)+(l*l)
B1

D.18 Binário:
512 256 128 64 32 16 8 4 2 1 256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1) 110100001
Complemento de um: **001011110** Complemento de dois: **001011111**
Verificação: O número binário original + seu complemento de dois
110100001 001011111
000000000

D.19 Zero.

Exercícios

D.20 Algumas pessoas argumentam que muríos de nossos cálculos seriam mais fáceis no sistema de numeração de base **12** porque **12** é divisível por muito mais números do que **10** (para base **10).** Qual o menor dígito na base **12** ? Qual pode ser o maior símbolo para o dígito da base **12** ? Quais os valores posicionais das quatro posições da direita de qualquer número no sistema de numeração da base **12?**

D.21 Como o maior valor de símbolo nos sistemas de numeração analisados se relaciona com o valor posicional do primeiro dígito à esquerda do dígito mais à direita de qualquer número nesses sistemas de numeração?

D.22 Complete a tabela de valores posicionais a seguir para as quatro posições da direita em cada um dos sistemas de numeração indicados:
decimal **1000 100 10 1**
base 6 **... ... 6 ...**
base 13 ... **169 ... ...**
base 3 **27 ... ...**

D.23 Converta o binário **100101111010** em octal e em hexadecimal.

D.24 Converta o hexadecimal **3A7D** em binário.

D.25 Converta o hexadecimal **765F** em octal. (Sugestão: Converta inicialmente **765F** em binário e depois converta aquele número binário em octal.)

D.26 Converta o binário **1011110** em decimal.

D.27 Converta o octal **42 6** em decimal.

D.28 Converta o hexadecimal **FFFF** em decimal.

D.29 Converta o decimal 299 em binário, em octal e em hexadecimal.

D.30 Mostre a representação binária do decimal **779.** Depois mostre os complementos de um e de dois de **779.**

D.31 Qual o resultado da soma do complemento de dois de um número com o próprio número?

D.32 Mostre o complemento de dois do valor inteiro **-1** em um equipamento com inteiros de 32 bits.